AFRL-RI-RS-TR-2015-217

GOFFISH: GRAPH-ORIENTED FRAMEWORK FOR FORESIGHT AND INSIGHT USING SCALABLE HEURISTICS

UNIVERSITY OF SOUTHERN CALIFORNIA

SEPTEMBER 2015

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2015-217 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
NANCY ROBERTS
Work Unit Manager

/ S /
MICHAEL J. WESSING
Deputy Chief, Information Intelligence
Systems and Analysis Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
SEPTEMBER 2015	FINAL TECHNICAL REPORT	SEP 2012 – JUN 2015

4. TITLE AND SUBTITLE

GOFFISH: GRAPH-ORIENTED FRAMEWORK FOR FORESIGHT AND INSIGHT USING SCALABLE HEURISTICS

5a. CONTRACT NUMBER
FA8750-12-2-0319

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Viktor Prasanna

5d. PROJECT NUMBER
XDAT

5e. TASK NUMBER
A0

5f. WORK UNIT NUMBER
16

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California
PO Box 3707, M/S 4C-76
Seattle, WA 98124-2207

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIEA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2015-217

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2015-4324
Date Cleared: 16 SEP 2015

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In this project, designed, developed, and built **GoFFish,** which is a scalable platform for graph-oriented event analytics that accelerates a wide class of DoD algorithms significantly. GoFFish is designed to be a scalable, modular, integrated, open-source analytics platform to fish for insight from massive event reservoirs archived from interconnected sensor event streams. The experiments performed under this effort showed speedups for certain classes of graph algorithms and also for their parallel algorithms for Louvain community detection and high betweenness centrality. This final report summarizes the GoFFish framework, and several parallel algorithms for graph processing.

15. SUBJECT TERMS

Graph event analytics, cloud computing, big data analytics, open source analytics

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**NANCY ROBERTS**
U	U	U	UU	128	19b. TELEPHONE NUMBER (*Include area code)* **315-330-3566**

Table of Contents

List of Figures

1. SUMMARY

In this project we designed, developed, and built **GoFFish,** which is a scalable platform for graph-oriented event analytics that accelerates a wide class of DoD algorithms significantly. GoFFish is designed to be a scalable, modular, integrated, open-source analytics platform to fish for insight from massive event reservoirs archived from interconnected sensor event streams. GoFFish consists of **GoFS,** a graph-oriented distributed event reservoir that uses intelligent data layout heuristics to store trillions of events. This minimizes the I/O complexity for distributed data access and subsetting, which significantly mitigates the effect of disk I/O bottleneck on big data analytics. The layout uses automated coalescing, partitioning and replication strategies for *targeted utilization of the inherent structure* present between the complex and heterogeneous event sources. A flexible query API allows efficient subsetting and extraction of datasets as *semi-structured events or subgraphs*, in parallel from a GoFS deployment on a commodity cluster or virtual machines in a Cloud computing environment. This platform was used on several graph datasets provided by XDATA to show speedups in storing and accessing data. We developed several graph algorithms and executed them with these datasets on cluster computers to demonstrate significant speedups.

In addition to GoFS we designed a programming abstraction called **Gopher** which is tailored for subgraph centric programming and targets the deficiencies of vertex centric approaches. Gopher sees graphs not as a collection of vertices but as a set of subgraphs defined as weakly connected components and which can be processed on independent machines. Each sub-graph is treated as an independent unit of computation within a Bulk Synchronous Parallel superstep. This effectively replaces the vertex as the basic computation unit from the classic vertex unit and allows a greater flexibility by performing more operations localized and reducing the communication overhead.

Our experiments demonstrated up to 81x speedups for certain classes of graph algorithms and datasets. They have showed that the subgraph centric approach is ideal for algorithms that require clustering and not suited for cases where there is a lot of per vertex computations.

Table 1. GoFFish improvement compared with the vertex centric approach

	Connected component	Single Source Shortest Path	Page Rank
California Road Network	81x improvement	32x improvement	4x improvement
Trace route path network of a CDN	21x improvement	10x improvement	1.5x improvement

In addition, our parallel algorithms for Louvain community detection and high betweenness centrality have shown speedups of 6x and 12x respectively, when compared to sequential and state–of-the-art approaches, and exhibited low degradation in the quality of the results.

We have successfully demonstrated our file system and its use with graph datasets and our algorithms at DARPA summer camps. This final report describes the main scientific and technological achievements between September 17, 2012 and June 10, 2015. It includes details on the GoFFish framework, the Hatrick framework for online event detection in evolving graphs, and several parallel algorithms for graph processing. The meetings with the PM, XDATA performers, as well as summer camp activities are also summarized. All published papers are attached in the appendix.

The following personnel were supported during this execution of the project:

Viktor Prasanna, PI

Cauligi Raghavendra, Co-PI

Yogesh Simmhan, Co-PI

Marc Frincu, Postdoctoral research associate

Alok Kumbhare, Ph.D. student

Charith Wickramaarachchi, Ph.D. student

Soonil Nagarkar, Graduate student

Santosh Ravi, Graduate student

2. INTRODUCTION

The ability to perform high fidelity observations through online instruments and sensors, and archive them on cheap storage is contributing to the growing **big data analytics challenge**. These datasets are unique in that they represent events, **observations and activities** that are **related** to one another while being recorded independently. **Analytics** over these massive event datasets should recognize the interconnected nature of events and their sources, rather than treating them as a flat table of tuples. Such knowledge is essential both to perform effective pattern mining and for efficient storage and access to data for efficient processing.

For **example**, a collection of urban surveillance equipment may be deployed outside buildings in a region to record environmental features such as vehicle and people. The time series of events generated by each equipment may list an object and its characteristic, such as <eqp1, car, red>, and be archived for analysis. These devices themselves are interconnected, in their spatial distribution and the features they observe. We can treat the devices as nodes in a graph, their spatial distribution as edges, and the events they generate as instances of this graph. One interesting data mining problem may be to find vehicles that follow a certain route near a soft target. Taking a graph-oriented view of event stream reduces this to a sub-graph clustering problem that **detects in real-time events** of a particular type (<car, red>) that are observed by specific devices (eqp1, … eqp8 that are nodes in the graph) in a particular order (eqp2 → eqp3 → eqp5 → eqp6) over a period of time.

The previous example outlines one of the defining characteristic of complexity in Big Data, i.e., its intrinsic interconnectedness, endemic to novel applications in both the Internet of Things and Social Networks. Such graph datasets offer unique challenges to scalable analysis, even as they are becoming pervasive. There has been significant work on parallel algorithms and frameworks for large graph applications on HPC clusters, massively multi-threaded architectures, and GPUs. Our focus in this project, however, was on leveraging commodity hardware for scaling graph analytics. Such distributed infrastructures, including Clouds, have democratized resource access, as evidenced by popular programming frameworks like MapReduce. While MapReduce's tuple-based approach is ill-suited for many graph applications, recent vertex-centric programming abstractions, like Google's Pregel and its open-source version, Apache Giraph, marry the ease of specifying a uniform logic for each vertex with a Bulk Synchronous Parallel (BSP) execution model to scale. Independent vertex executions in a distributed environment are interleaved with synchronized message exchanges across them to form iterative supersteps. However, there are short-comings to this approach.

(1) Defining an individual vertex's logic forces costly messaging even within vertices in one partition.
(2) Porting shared memory graph algorithms to efficient vertex centric ones can be non-trivial.
(3) The programming abstraction is decoupled from the data layout on disk, causing I/O penalties at initialization and runtime. A recent work identified the opportunity of leveraging shared memory algorithms within a partition, but relaxed the programming model to operate on a whole partition, without guarantees of sub-graph connectivity within a partition or implicit use of concurrency across sub-graphs.

Our research on graph-oriented event analytics transforms traditional approaches. Our software stack leverages this knowledge for data layout, programming models, and algorithmic advances to offer an order of magnitude improvement. **GoFFish stack** comprises of:

- **GoFS** distributed file system
- **Gopher** programming model
- **Floe** stream processing engine for fast streaming data analytics, and
- **Hatrick** graph abstraction that enables massive graph-oriented analytics over large event reservoirs.

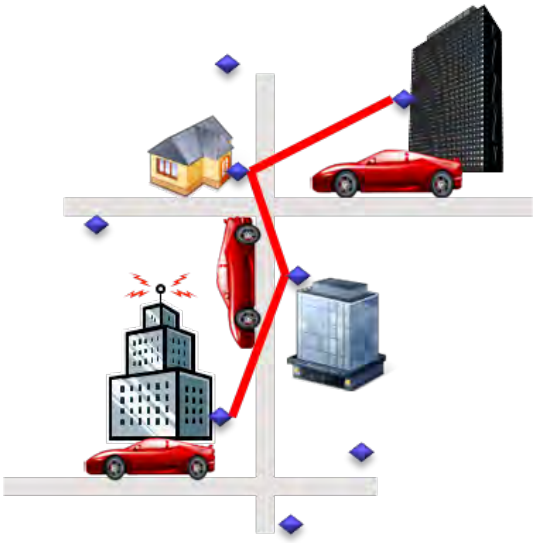


Figure 1. Example of a graph based application.

GoFFish is **modular**, and each sub-stack bottom-up can be used by itself as part of a **Big Data analytics** eco-system. Consequently, our research work provided frequent and composable components that can be used and integrated by other performers and end users.

In addition to developing and testing the GoFFish stack we focused on enhancing existing algorithms for graph analytics. We addressed two algorithms used by XDATA performers, namely the Louvain community detection and the high betweenness centrality detection.

Detecting community structures in graphs is a well-studied problem in graph data analytics. Unprecedented growth in graph structured data due to the development of the World Wide Web and social networks in the past decade emphasizes the need for fast graph data analytics techniques. We designed and tested a simple yet efficient approach to detect communities in large scale graphs by modifying the sequential **Louvain algorithm** for community detection. The proposed distributed memory parallel algorithm targeted the costly first iteration of the initial method by parallelizing it.

Betweenness centrality (BC) is an important measure for identifying high value or critical vertices in graphs, in variety of domains such as communication networks, road networks, and social graphs. However, calculating betweenness values is prohibitively expensive and, more often, domain experts are interested only in the vertices with the highest centrality values. Our approach was to first propose a partition-centric algorithm (MS-BC) to calculate BC for a large distributed graph that optimizes resource utilization and improves overall performance. Then we extended the notion of approximate BC by pruning the graph and removing a subset of edges and vertices that contribute the least to the betweenness values of other vertices (MSL-BC), which further improved the runtime performance.

We evaluated the two proposed algorithms on real-life datasets and emphasized their strengths and weaknesses through benchmarks performed on the local cluster and the USC Center for High Performance Computing.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

The results presented in this final report were facilitated by the expertise of the PIs and senior personnel which had weekly meetings with students where progress on both technical and scientific tasks was discussed. In addition, team members have participated in XDATA summer camps and demonstrations where they presented their findings and interacted with other XDATA performers to ensure that the team's objectives are aligned with the overall program scope and that the results can be integrated in software products developed by other teams. The PIs have interacted with the PM to adjust the research objectives and to refocus the project at the end of year 1.

Our initial focus was the development of the GoFFish framework capable of large scale graph analytics on clouds. After our discussions with the PM we have refocused on parallel algorithms and software for streaming graph analysis.

Assumptions. The GoFFish framework for cloud oriented graph analytics relies on the Bulk Synchronous Parallel (BSP) model which allows algorithms to be executed in parallel on graph partitions but to synchronize at periodic intervals. This can induce delays in execution if the partitioning is not balanced from a computational point of view. In our approach we have assumed a minimum cut partitioning technique where the graph is partitioned by trying to minimize the edge cuts between partitions. In addition, our initial GoFFish assumption was that graphs remain static in terms of number of vertices but can change the edges connecting them.

Regarding the initial version of our online event detection on evolving graphs (i.e., Hatrick) we have considered in our algorithms only edge additions due to the complexity of handling edge removals.

Due to refocusing our activities, in the second year we investigated methods for speeding up graph processing by parallelizing existing algorithms. Our approach was to use a distributed memory approach on parallel machines, making our algorithms adjustable for distributed computing systems such as GoFFish or Apache Giraph.

Procedures. Our GoFFish and Hatrick frameworks, and the proposed algorithms were tested on the group's cloud enabled cluster and on the USC's Center for High-Performance Computing. Tests were performed on XDATA datasets (e.g., Akamai, Twitter) as well as publicly available datasets (e.g., California Road Network - CRN, LiveJournal - LJ).

4. RESULTS AND DISCUSSION

Our main results consist of research papers, software products, and demos.

We have published **13 papers** out of which 9 conference articles, 1 journal paper, 2 technical reports, and 1 poster:

- 3 papers are focused on the GoFFish platform
- 2 papers are focused on continuous dataflows and real-time analytics models.
- 6 papers are focused on efficient resource management algorithms and approaches for streaming data on elastic clouds
- 2 papers are focused on parallelizing graph algorithms for speeding up graph oriented analytics

The full publication list can be found in the reference list and the papers are attached at the end of this report. The main results are summarized and discussed below.

4.1. GoFFish platform

GoFFish architecture provides a modular and vertically integrated framework stack for graph-oriented analytics over massive event reservoirs [1, 8]. The modules in the stack are **GoFS** for distributed storage of event reservoirs, **Gopher** for programming and scalable execution of graph and event analysis over event reservoirs in GoFS. *Each module is designed to be usable independently of the ones above it in the stack (e.g. by other performers).*

Our architecture is predicated on running within a distributed cluster or Infrastructure-as-a-Service Cloud environment. At the bottom of the stack, the **Graph-Oriented File System (GoFS)** provides the storage layer for intelligent layout of the event reservoir across a distributed cluster. GoFS offers a distributed file system that works on top of the existing file system present on individual nodes of the cluster. Events from a single source or of a particular type are called a *Pool* and contain a series of time-stamped and uniquely identified event tuples each of which is a bag of keys and values. Event tuples in a pool are schema-free. An event *reservoir* is a collection of event pools with explicit named relationships between them that can be specified by a user with limited knowledge about the pools and their usage scenarios. GoFS treats a reservoir as a multi-graph, with pools serving as vertices and their named relationships as edges. *GoFS uses*

heuristics based on temporal and edge distances, and key names to group, partition and replicate the event pools in a reservoir into buckets, with each bucket being stored as a file in the file system. GoFS maintains an index file for the reservoir with metadata for the buckets' contents and their locations within the cluster. Event pools for a reservoir are bulk loaded through API's provided by GoFS and online partitioning, grouping and replication is performed during the insert. APIs are also provided for extracting buckets from the closest replica and recombining them into data subsets that are graph or event-centric based on the application needs.

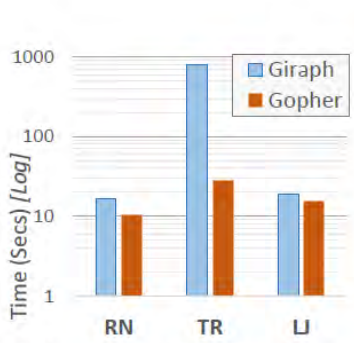
Graph oriented programing on event reservoirs (Gopher) represents the dataflow programming model for rapid composition of graph and event based analytics executed at scale on distributed clusters that leverages extraction APIs provided by GoFS. It offers a multi-stage pipeline composition familiar to Iterative MapReduce and Bulk-Synchronous Parallel (BSP) approaches, but with *more powerful data models* that include subgraphs and temporal event subsets, besides vertex-centric messages (e.g., Pregel) and key-value pairs (e.g., MapReduce). Its **subgraph centric approach** allows GoFFish to harness the connected component structures found in large graph types and to reduce the network communication overhead of classic vertex centric models (e.g., Pregel, Giraph). Each stage performs a single application logic but is executed by the framework across different nodes in the cluster and over different data subsets extracted and recomposed by GoFS or transferred from a previous stage while preserving data locality. The output from each stage is efficiently streamed to the next based on graph-vertex grouping, temporal region or hash over a key, and disk I/O for the intermediate dataflow minimized using aggregation techniques. This provides an intuitive yet power model for efficient graph and event based analysis at large scales.

Experimental results comparing GoFFish with Giraph demonstrate the advantage of the subgraph centric model for certain classes of algorithms (see Figure 2).

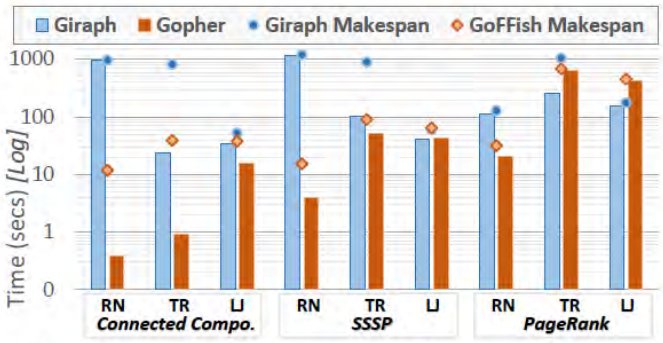
The original GoFFish was only capable of performing analytics on static graphs. However, since modern large scale graphs generated by interconnected sensor networks evolve over time, we have **augmented** its interface with **support for time-series graph analysis** [12]. For example, consider a road network in a Smart City. The road topology remains relatively static over days. However, the traffic volume monitored on each road segment changes significantly throughout the day, as do the actual vehicles that are captured by traffic cameras. Widespread urban monitoring systems, community mapping applications, and the advent of self-driving cars will continue to enhance our ability to rapidly capture changing road information for both real-time and offline analytics. To enable time series analysis we have improved GoFFish's support for handling graph instances (i.e., graph snapshots taken at fixed intervals). Based on these instances we then perform incremental graph analytics. For this we identified three **design patterns** for temporal graph algorithms (see Figure 3):

1. Analysis over every graph instance is **independent**. The result from the application is just a union of results from each graph instance;
2. Graph instances are **eventually dependent**. Each instance can execute independently but results from all instances are aggregated in a Merge step to produce the final result;
3. Graph instances are **sequentially dependent** over time. Here, analysis over a graph instance cannot start (or, alternatively, complete) before the results from the previous graph instance are available.

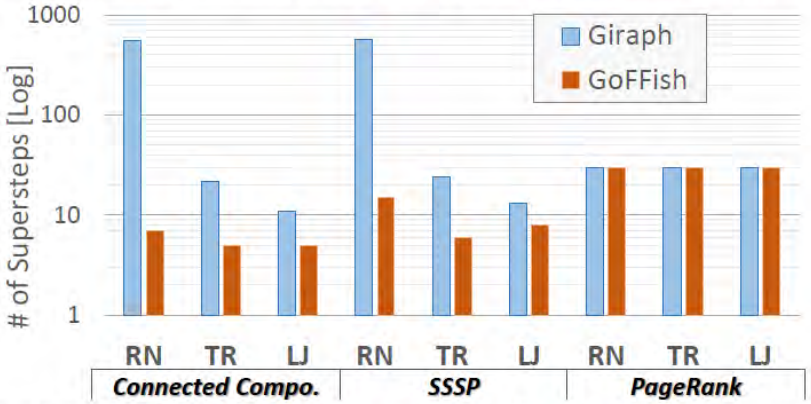
Experimental results, performed on three representative applications (e.g., time dependent shortest path – TDSP, meme tracking, and hashtag aggregation) from each of the identified pattern class have shown the scalability of our approach (see Figure 4).



(a) Graph loading time (log scale) from disk to memory.



(b) Compute time and Makespan (incl. load) for GoFFish and Giraph for various graph algorithms and datasets (log scale)



(c) Number of supersteps (log scale) taken by Gopher and Giraph for various graph algorithms and datasets

Figure 2. Comparison of GoFFish and Giraph for Connected Components, Single Shortest Path and Page Rank algorithms.

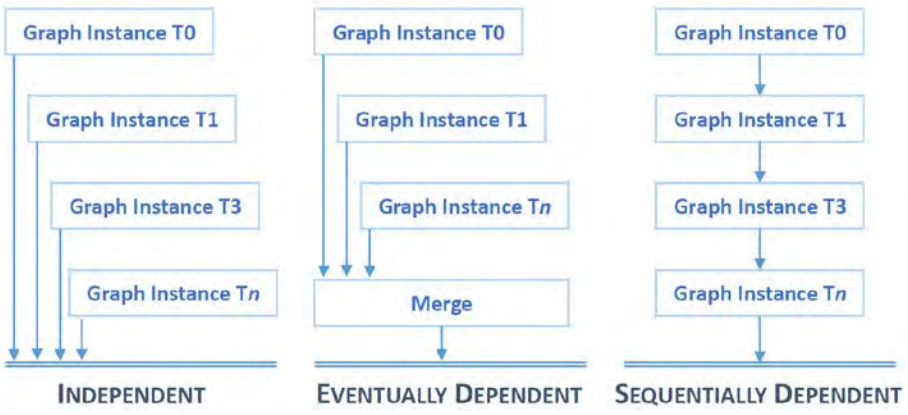
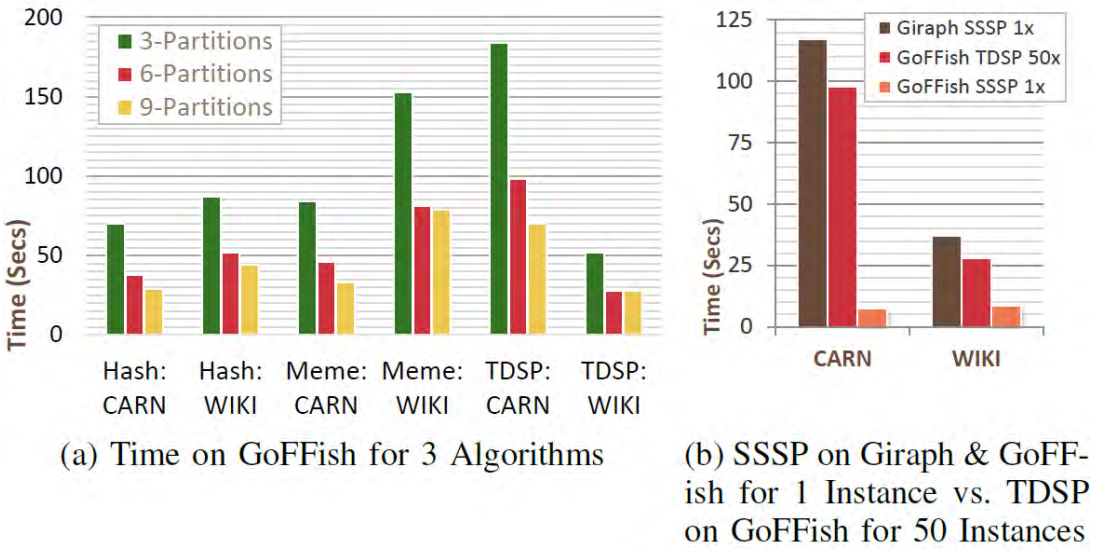


Figure 3. Design patterns for time-series graph algorithms.



(a) Time on GoFFish for 3 Algorithms

(b) SSSP on Giraph & GoFFish for 1 Instance vs. TDSP on GoFFish for 50 Instances

Figure 4. Time taken by different algorithms on time-series graph datasets.

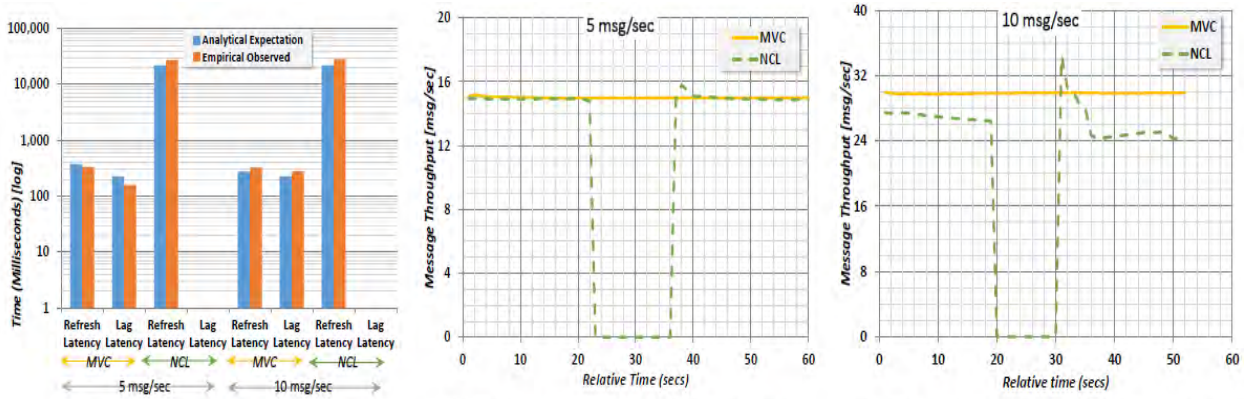
4.2. DataFlows and real-time analytics on streaming graphs

While time series analysis on graph instances is an important step towards evolving graph analysis, much of the graph data generated is streamed at high rates. Capturing key events in this Big Data deluge requires **low latency analytics** especially for critical mission areas such as detecting threats from social network interactions and computer network breaches. We focused on two key aspects: 1) continuous dataflow update strategies in case of incremental updates or bug fixes [2], and 2) enabling real-time analytics on streaming graph [4].

Continuous dataflows complement scientific workflows by allowing composition of real-time data ingest and analytics pipelines to process data streams from pervasive sensors and "always-on" scientific instruments. Such dataflows are mission critical applications that cannot suffer downtime, need to operate consistently, and are long running, but may need to be updated to fix bugs or add features. This poses the problem: *How do we update the continuous dataflow application with minimal disruption?* To answer this question we formalized different types of **dataflow update models** (processor update, channel update, independent subgraph update, and connected subgraph update) for continuous dataflow applications, and identified the **qualitative and quantitative metrics** (consistency, interleaved and delineated, refresh latency, lag latency, throughput, and message loss) to be considered when choosing an update strategy. Finally we proposed **five dataflow update strategies**, and analytically characterized their performance trade-offs:

1. Naïve Consistent Lossy Update (NCL)
2. Naïve Consistent High-latency Update (NCH)
3. High-Throughput Inconsistent Update (HTI)
4. Message-Versioned Consistent Update (MVC)
5. Path-Versioned Consistent Update (PVC)

We implemented MVC update strategy for the processor update type within our Floe continuous dataflow engine and compared it against the NCL update strategy. Figure 5 shows the results.



(a) Refresh and Lag Latencies (b) Observed Throughput (5 msg/sec input) (c) Observed Throughput (10 msg/sec input)

Figure 5. Experimental and/or Analytical latency and throughput values when performing Message-Versioned Consistent Update (MVC) and Naive Consistent Lossy Update (NCL) on USC Microgrid dataflow with input rates of 5, 10 msg/sec.

Next we focused on enabling real-time analysis for large scale streaming graphs such as social networks. While making sure that dataflow updates do not impact the processing rate, there is also the need to devise a suitable model for incremental graph analytics on streaming graphs. In [4] we propose a preliminary model based on graph instances which allows for low latency detection of graph events, and analyze two use cases from the area of smart grids and network security. Starting from this model we have built the Hatrick framework for incremental graph analytics. It is built on top of our Floe stream processing engine and allows **low latency distributed graph analytics** following a BSP-like programming model and a simple user API for updating the graph and processing incoming messages. The system architecture is depicted in Figure 6.

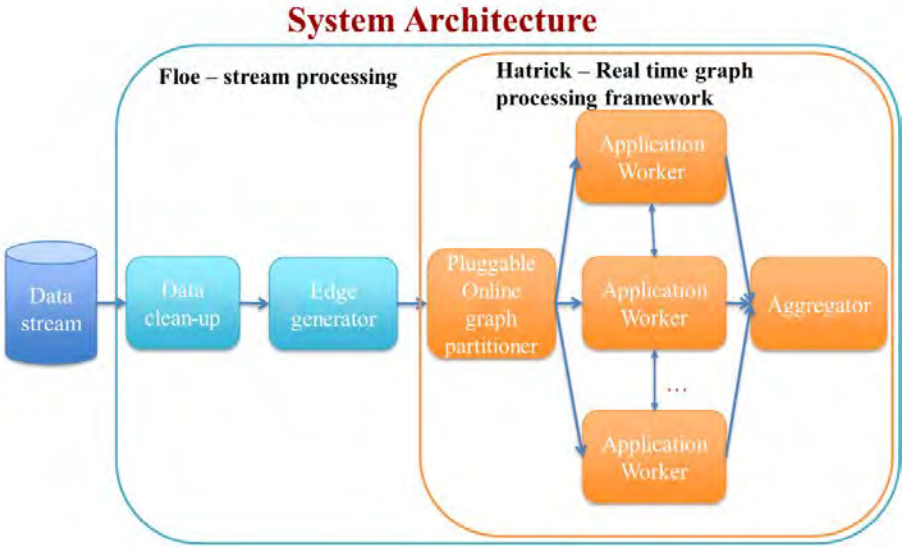


Figure 6. Hatrick framework on top of Floe.

4.3. Efficient resource management for streaming data on elastic clouds

A key issue when dealing with low latency processing of streaming (graph) data is that of efficiently allocating resources and task scheduling.

To deal with variations in cloud performance and data throughput rates we first introduced the concept of **dynamic dataflows** which allows us to switch at runtime the tasks that process data [3]. This replacement with other similar tasks (Processing Elements – PEs) that are capable of different performance allows us to deal with performance fluctuations differently than traditional horizontal and vertical elasticity of cloud resources. We formalized an optimization problem to perform both deployment and runtime cloud resource management for such dataflows, and defined an objective function that allows trade-offs between the application's value against resource cost. Finally we presented two **novel reactive heuristics**, local and global, based on the variable sized bin packing heuristics to solve this NP-hard problem. The local strategy calculates an alternate's cost based only on its processing requirement. The global strategy calculates the cost of the alternate as the sum of its processing needs and that of its downstream PEs.

Our experimental results show that the continuous re-deployment heuristic which makes use of application dynamism can reduce the execution cost by up to 15% on clouds while also meeting the QoS constraints.

Reactive heuristics have the drawback of increasing the adaptation time of the dataflow which is important in mission critical scenarios. To deal with this issue we proposed in [5] several **pro-active methods** including PLAStiCC. The concept behind this algorithm is described as follows. With a perfect oracle, the predicted values for an interval remain fixed for that interval. However, given the limitations of state-of-the-art multivariate time series models, in practice, the errors in the predicted values will increase with the horizon of prediction. Hence, we will get better prediction values as we move further into a prediction interval. To utilize this, we allow new predictions to be made at each timestep rather than at the end of each interval. Further, anytime a new (improved) prediction changes existing predicted values, we start a new interval and replan actions for all its timesteps, replacing the prior plan. This sliding window of replanning does not change the optimization problem, except that the optimization decisions planed in one interval may be altered in the succeeding interval if updated predictions are available. However, note that this is different from the reversal of decisions in the reactive version, since the decisions there are enacted immediately for the next timestep whereas in the look-ahead version the replanning may just change a future action and hence not incur resource penalty.

We have performed extensive tests [11] on both our reactive and pro-active algorithms and compared them with a genetic algorithm known for being highly efficient in scheduling problems due to its intrinsic capability of exploring a wide range of solutions. Results are depicted in Figure 7.

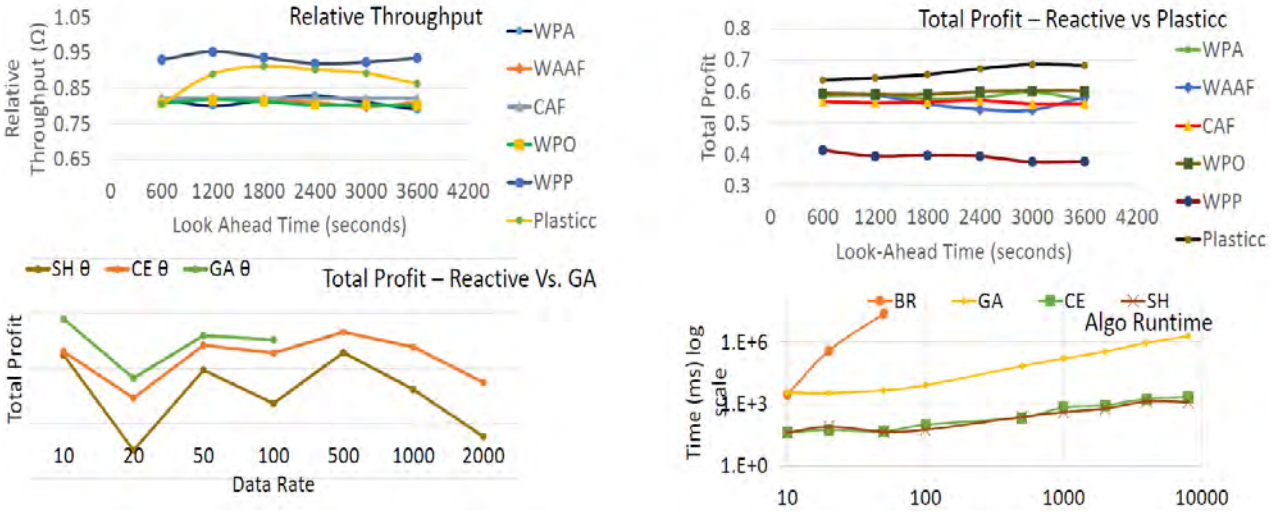


Figure 7. Comparison between our Reactive, PLAStiCC, and the genetic algorithm.

Overall the following **conclusions** can be drawn:

- Dynamic dataflows provide powerful abstractions and flexible runtime adaptation
- Reactive Scheduling Techniques address Infrastructure Performance and Data Rate variations but are prone to thrashing
- Predictive Look-Ahead Scheduling generates an evolving plan and mitigates thrashing
- Genetic Algorithms provide near optimal solutions but have high run-time overheads, and hence are suitable only for initial deployment

Results were also presented during the IPDPS 2014 Ph.D. Forum and Poster session [7].

Fast **elasticity**, **load balancing**, and **fault-tolerance** are also key aspects that need to be considered by any streaming graph processing engine. These features are tightly linked to the problem of efficient resource management. Elasticity requires careful planning, fast allocation of resources at runtime, and efficient scheduling of dataflow tasks such that the desired throughput is satisfied. Load balancing allows us to optimally use resources and to avoid under and over provisioning. Finally, fault-tolerance requires intelligent data placement near existing tasks to minimize latency in case of failures. In [13] we proposed a model for fault-tolerant and elastic streaming MapReduce model and validated it on our Floe platform. This model is based on a consistent hashing approach and has the advantage of offering a unified view of all three aspects: elasticity, load balancing, and fault-tolerance.

Experimental results (see Figure 8) comparing our approach with Apache Storm show the superior throughput our solution achieves.

Finally, to mitigate the cost of running cloud based graph analytics we investigated the potential to **leverage** the availability of cheaper, but less reliable, **spot VMs** from cloud providers to reduce the cost of hosting cloud applications [6]. Specifically we proposed scheduling techniques where we mix on-demand fixed priced VMs with spot instances. The difficulty here is that spot instances' prices fluctuate and are highly unreliable as they can be terminated without notice at any point during their lifecycle. We proposed strategies to manage a job's life-cycle on spot and on-demand VMs to minimize the total dollar cost while assuring completion. With the foundation of stochastic optimization, our reusable table-based algorithm (RTBA) decides when

to instantiate VMs, at what bid prices, when to use local machines, and when to checkpoint and migrate the job between these resources, with the goal of completing the job on time and with the minimum cost. In addition, we proposed three simpler heuristics (H1, H2, H3) for comparison.

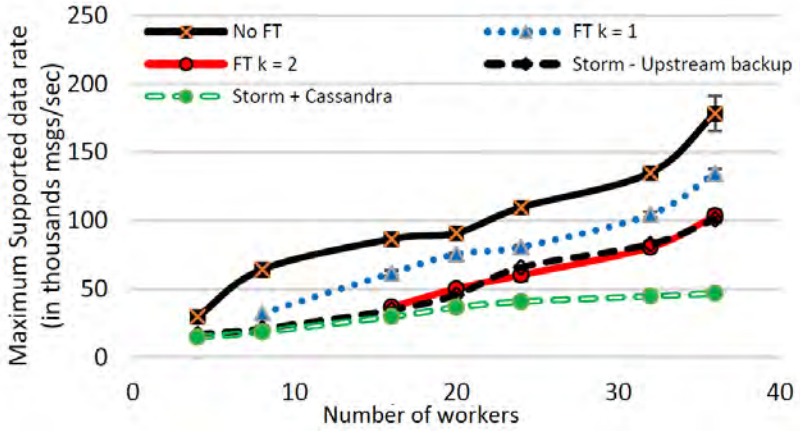


Figure 8. Throughput of our proposed fault-tolerant and elastic Map Reduce on Floe and Apache Storm.

Our evaluation using historical spot prices for the Amazon EC2 market showed that RTBA on an average reduces the cost by 72%, compared to running only on on-demand VMs. It also showed it to be robust to fluctuations in spot prices. The heuristic, H3 had similar performances as RTBA and may prove adequate for ad hoc jobs due to its simplicity.

4.4. Parallel algorithms

In the second year of phase 1 we have shifted and refocused our efforts on speeding important graph analytics algorithms for use by the XDATA community using the concepts developed earlier. This refocus was done in consultation with DARPA PM in February 2014 to bring valuable fast parallel algorithms for important graph analytics. We initiated collaborations with Sotera and identified several analytics kernels and datasets of importance to their analysis. Starting from our initial discussions we have designed, implemented, and benchmarked two kernels: **Louvain community detection** and **approximate high betweenness centrality** extraction.

Louvain community. Large graphs exhibit patterns that can be viewed as fairly independent compartments with distinct roles, i.e., communities that can be identified based on the graph structure or data clustering. The advent of social networking and online marketing poses new challenges for community detection due to the large data size which can reach up to millions and even billions of vertices and edges. In addition to the graph size, the advent of cloud computing brought the reality of distributed big data to the picture. To cope with this scenario parallel and distributed community discovery algorithms need to be designed. This requires the graph to be initially partitioned across processors so that communication between them during graph

processing is minimized. The dynamic evolving nature of these graphs represents another main challenge that emphasizes the need for fast graph analytics. Our approach to solving these challenges was to take advantage of the initial graph partitioning when performing parallel community detection in order to speed-up the process by minimizing the communication between processors. It thus takes advantage of the subgraph centric model. The design of several graph partitioning algorithms worked in favor of this idea as they try to minimize the cross partition edges between partitions. This reduced the chance for communities based on graph structural information to be spread across multiple partitions. We demonstrated our approach by implementing an MPI version of the Louvain community detection algorithm.

Betweenness centrality. With the development of massive online social networks and graph structured data in various areas such as biology, transportation, and computer networks, analyzing the relationship between the various entities forming the network is increasingly important as it allows complex behavior in these large graphs to be explained. The relative location of each vertex in the network helps identify the main connectors, where the communities are, and who is at their core. Centrality indices measure the importance of a vertex in a network based on their position in the network. Betweenness centrality (BC) is one such useful index that is based on the number of shortest paths that pass through each vertex. Our approach was to propose a solution that takes advantage of the subgraph centric approach by proposing a partition-centric bulk synchronous parallel algorithm that exhibits more efficient utilization of distributed resources and at the same time shows significant reduction in the number of supersteps as compared with the vertex centric approaches. We demonstrated our approach by implementing an MPI version of the approximative high betweenness centrality algorithm.

One of the main advantages of our MPI based approaches is that they allow reusing single node sequential algorithms in parallel implementations. These were tested on several synthetic and real-life graph datasets on a local HPC cluster at USC.

In the case of the Louvain algorithm we took a novel approach by combining graph partitioning with community detection in order to make the execution embarrassingly parallel. Figure 9 shows the speed-up results for the algorithm on 16-node cluster with 128 cores. Benchmarks have shown that without any noticeable degradation in the quality of the communities we can achieve speed-ups of up to 6 times for synthetically generated community graphs.

For the approximate high betweenness centrality detection algorithm we designed a novel algorithm which parallelizes the single shortest path computations from each subgraph to reduce execution time by improving resource utilization and reducing synchronization cost aggregated over all supersteps. In doing so we sacrificed memory usage while increasing parallelism. We also improve algorithm performance by pruning the graph using a recursive leaf-compression approach thus reducing the graph size and in turn eliminating unnecessary computations. Results depicted in Figure 10 show that our algorithm achieved speed-ups of up to 12x for sparse graphs. We have tested the algorithm on datasets including Enron, Gowalla, Pennsylvania road network, and several synthetically generated graphs. To check the performance of our algorithm on more challenging graphs we have further evaluated it on Graph 500 datasets which exhibit a power-law graph structure and show up to 2-3x speed-ups compared to traditional approaches.

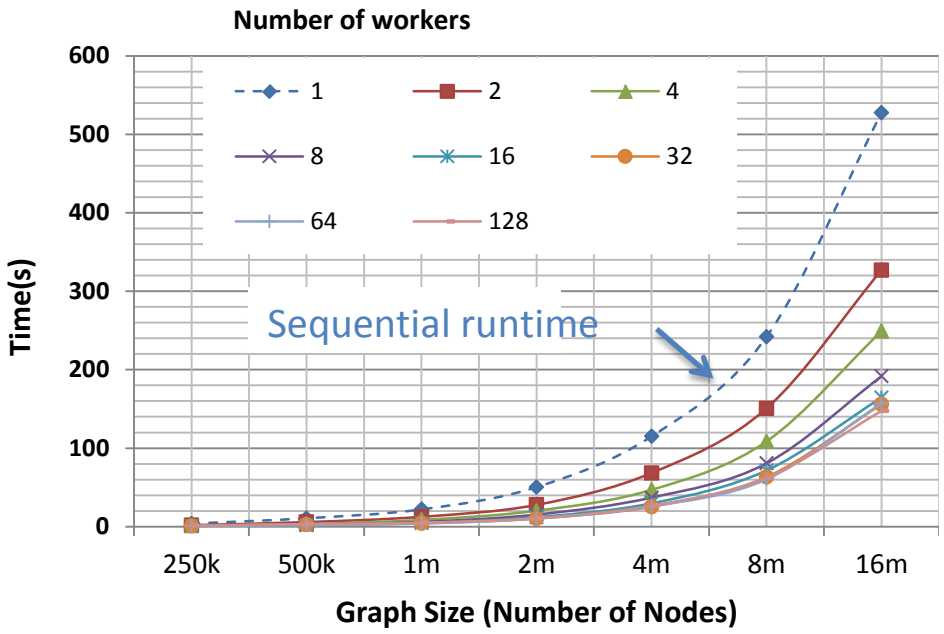


Figure 9. Speed-up of our Proposed Louvain algorithm.

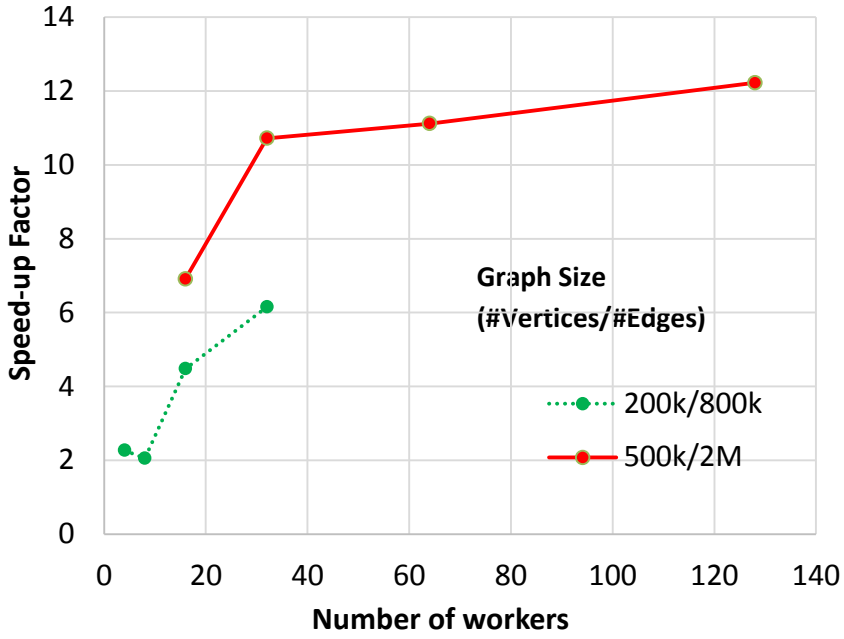


Figure 10. Speed-up of our proposed approximate high betweenness centrality algorithm.

4.5. Summer camps' outcome

We participated in two summer camps organized in 2013 and 2014, respectively.

During the 2013 summer camp we initiated collaborations with several XDATA teams including, **Sotera**, **Oculus**, **JPL**, **Kitware**, and **Indiana University**. We have performed experiments on numerous XDATA (**Akamai** and **Twitter**) and **non-XDATA datasets.** We worked with **Kitware** to produce a time series graph visualization tool and with **Indiana University** to enable abstract rendering and visualization of graph statistics.

During the 2014 summer camp we collaborated with **Sotera** (**Jonathan Larson**), **Oculus** (**Scott Langevin**) and **JPL** (**Lewis John McGibbney**) and focused our activities on three main directions:

1) Integrating the two analytics kernels developed following our interaction with **Sotera** in a visualization tool provided by **Oculus**;
2) Implementing a MapReduce version of our Louvain algorithm for **Sotera**;
3) Updating the XDATA open catalog with our latest products and documentation.

Our progress and planned activities were reported at the end of summer to the XDATA PM and also sent to Sotera for feedback. The report contained results on GoFFish, parallelization of Louvain and Betweenness Centrality, and proposed plan for incremental graph analytics on streaming graphs. The report was based on the presentation given to the PM during the summer camp at the PI meeting day.

4.5.1. Kernel integration in Aperture.

Most of the summer camp activities were focused on working with a team from **Oculus** to integrate our analytics kernels in their Aperture API to visualize central nodes in communities at various levels of detail, which we have completed and demonstrated at the end of the summer camp. An application screenshot is shown in Figure 10. The dataset we used was the **South American Twitter snapshot**. By relying on our fast kernels Aperture is able to display the top centrality nodes inside different levels of Louvain communities. This is important in identifying the most influential users and the communities they belong to. By clicking on nodes application users can also view the Twitter users' profiles for further details. The demo was made available inside the XDATA network at http://10.1.93.218:8080/twitter-community-demo/.

Figure 11. Centrality nodes inside communities using Aperture.

4.5.2. MapReduce based Louvain algorithm.

Our algorithms have been implemented using MPI and while extremely efficient they are not currently supported by the XDATA stack. For this reason we have discussed with Sotera and decided to deliver a MapReduce version of the Louvain algorithm to Sotera. This was possible due to the embarrassingly parallel algorithm we designed. The MapReduce version can be directly integrated with Sotera's Apache Hadoop installation. By the end of the summer camp we have delivered to Sotera a version for them to test on their infrastructure against their Giraph based version. Several rounds of discussions on fixing issues and algorithm design were established.

4.6. Software products

We have published three software products in the XDATA open source catalog (http://www.darpa.mil/OpenCatalog/XDATA.html):

- The GoFFish platform
- Parallel MPI implementation for the Louvain modularity detection
- Parallel MPI implementation for the Betweenness Centrality detection

The documentation for each product has been updated with an **executive summary**, a **quick start guide**, and a **detailed documentation**. The quick start guide is intended for fast product evaluation on a single node machine while the detailed documentation shows the details of setting up and deploying the products on a distributed infrastructure. Each product comes with a ready to be tested VM for minimal setup overhead.

To maximize the impact of GoFFish, we have initiated a discussion group with the intent of building a community around the project that can support further development and maintenance. The group can be found at https://groups.google.com/forum/?hl=en#!forum/usc-cloud.

Additionally, our Floe streaming engine which is the basis for both GoFFish and Hatrick can be found at:

- https://github.com/usc-cloud/floe2

4.7. Demos

We have published a demo showing the use of our Hatrick framework for incremental graph analytics:

- http://tsangpo.usc.edu/realtimegraph

5. CONCLUSIONS

We have designed, developed and tested efficient frameworks and algorithms which were validated against real datasets.

- **GoFFish**: up to 81x speedups for certain classes of graph algorithms and datasets.
- **Parallel algorithms**:
  - **Louvain**: 6x improvement against sequential algorithms with no degradation in the accuracy metrics.
  - **Betweenness centrality**: 12x improvements for sparse graphs compared with state of the art solutions.
- **Hatrick**: first graph analytics platform for detecting events on evolving graphs and based on a distributed memory.

We have demonstrated our results by integrating them during summer camps with other performers' software stacks. Our software is publicly available on the XDATA website as well as on well-known public repositories.

We have disseminated our results in numerous international conferences and journals.

6. REFERENCES

[1] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, N. Ma, S. Nagarkar, S. Ravi, C. Raghavendra and V. Prasanna, **GoFFish : A Framework for Distributed Analytics Over Timeseries Graphs**, USC Tech Report No. 13-936, 2013

[2] C. Wickramaarachchi and Y. Simmhan, **Continuous Dataflow Update Strategies for Mission-Critical Applications**, 9th IEEE International Conference on e-Science, October 2013

[3] A. Kumbhare, Y. Simmhan and V. K. Prasanna, **Exploiting Cloud Elasticity to Enhance the Value of Dynamic, Continuous Dataflows** , IEEE/ACM International Conference for High Performance Computing Networking, Storage, and Analysis (SC) , November 2013

[4] C. Wickramaarachchi, M. Frincu and V. Prasanna, **Enabling Real-time Pro-active Analytics on Streaming Graphs**, USC Tech Report No. 14-947, February 2014

[5] A. Kumbhare, Y. Simmhan and V. K. Prasanna, **PLAStiCC: Predictive Look-Ahead Scheduling for Continuous dataflows on Clouds**, IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), May 2014

[6] H. Chu and Y. Simmhan, **Cost-efficient and Resilient Job Life-cycle Management on Hybrid Clouds**, 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2014

[7] A. Kumbhare, Y. Simmhan and V. K. Prasanna, **Constraint-Driven Adaptive Scheduling for Dynamic Dataflows on Elastic Clouds** , International Parallel & Distributed Processing Symposium (IPDPS) , May 2014 (Phd Forum and Poster)

[8] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra and V. Prasanna, **GoFFish : A Sub-Graph Centric Framework for Large-Scale Graph Analytics**, EuroPar Conference, August 2014

[9] A. Kumbhare, M. Frincu, C. Raghavendra and V. Prasanna, **Efficient Extraction of High Centrality Vertices in Distributed Graphs**, 18th IEEE High Performance Extreme Computing Conference (HPEC), September 2014 (candidate for the best paper award)
[10] C. Wickramaarachchi, M. Frincu, P. Small and V. Prasanna, **Fast Parallel Algorithm for Unfolding of Communities in Large Graphs**, 18th IEEE High Performance Extreme Computing Conference (HPEC), September 2014
[11] A. Kumbhare, Y. Simmhan, M. Frincu and V. Prasanna, **Reactive Resource Provisioning Heuristics for Dynamic Dataflows on Cloud Infrastructure**, IEEE Transactions on Cloud Computing, January 2015
[12] Y. Simmhan, N. Choudhury, C. Wickramaarachchi, A. Kumbhare, M. Frincu, C. Raghavendra and V. Prasanna, **Distributed Programming over Time-series Graphs**, 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2015
[13] A. Kumbhare, M. Frincu, Y. Simmhan, V. Prasanna, **Fault-Tolerant and Elastic Streaming MapReduce with Decentralized Coordination**, 35th IEEE International Conference on Distributed Computing Systems (ICDCS), July 2015

List of Acronyms

BC: Betweenness Centrality
BSP: Bulk Synchronous Parallel
CARN: California Road Network
CDN: Content Delivery Network
FT: Fault Tolerance
GA: Genetic Algorithm
GoFFish: Graph-Oriented Framework for Foresight and Insight using Scalable Heuristics
GoFS: Graph Oriented File System
GPU: Graphics Processing Unit
HPC: High Performance Computing
HTI: High-Throughput Inconsistent Update
MPI: Message Passing Interface
LJ: Live Journal
MS-BC: Multi Source Betweenness Centrality
MSL-BC: Multi Source with Leaf Compression Betweenness Centrality
MVC: Message-Versioned Consistent Update
NCH: Naïve Consistent High-latency Update
NCL: Naïve Consistent Lossy Update
PLAStiCC: Predictive Look-Ahead Scheduling for Continuous dataflows on Clouds
PVC: Path-Versioned Consistent Update
RN: Road Network
SSSP: Single Source Shortest Path
TPDS: Time dependent shortest path
TR: TraceRoute
VM: Virtual Machine

APPENDIX – PUBLISHED PAPERS

Fast Parallel Algorithm For Unfolding Of Communities In Large Graphs

Charith Wickramaarachchi*, Marc Frincu†, Patrick Small* and Viktor K. Prasanna†
*Department of Computer Science
†Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, USA
Email: {cwickram, frincu, patrices, prasanna}@usc.edu

***Abstract*—Detecting community structures in graphs is a well studied problem in graph data analytics. Unprecedented growth in graph structured data due to the development of the world wide web and social networks in the past decade emphasizes the need for fast graph data analytics techniques. In this paper we present a simple yet efficient approach to detect communities in large scale graphs by modifying the sequential Louvain algorithm for community detection. The proposed distributed memory parallel algorithm targets the costly first iteration of the initial method by parallelizing it. Experimental results on a MPI setup with 128 parallel processes shows that up to ≈5x performance improvement is achieved as compared to the sequential version while not compromising the correctness of the final result.**

I. Introduction

Large graphs exhibit patterns that can be viewed as fairly independent compartments with distinct roles, i.e., communities that can be identified based on the graph structure or data clustering [1]. The advent of social networking and online marketing poses new challenges for community detection due to the large data size which can reach up to millions and even billions of vertices and edges. In addition to the graph size, the advent of cloud computing brought the reality of distributed big data to the picture. To cope with this scenario parallel and distributed community discovery algorithms need to be designed. This requires the graph to be initially partitioned across processors so that communication between them during graph processing is minimized. The dynamic evolving nature of these graphs represents another main challenge that emphasizes the need for fast graph analytics. Performing analytics on periodic graph snapshots is one of the proposed approaches to address this issue [2], [3]. However, due to the high velocities of data (e.g., Twitter reports more than 5,700 tweets per second on average [4]), performing analytics even on the snapshots needs to be completed fast for the results to be useful.

Graph partitioning and community detection can be seen as two sides of the same coin which try to achieve highly correlated goals. A major difference between the two is that for the former the number of partitions or partition size is known in advance. Most graph partitioning techniques were designed for parallel computing so that partitions are balanced among processors and communication between them is minimized [5]. Graph partitioning techniques are used in distributed graph analytics where they proved to give good performance for several classes of distributed graph algorithms [6][7]. For large graphs, partitioning is traditionally considered to be part of the data loading process to partition and load the graphs into the distributed storage. Graph algorithms and analytics are later performed on those partitions. Generally, partitioning to load balance and minimize edge cuts has been decoupled from the community detection with many algorithms using a random or hash based assignment of graph vertices to processors [8].

One of the main contributions of this work is to take advantage of the initial graph partitioning when performing parallel community detection in order to speed-up the process by minimizing the communication between processors. The design of several graph partitioning algorithms [9] work in favor of this idea as they try to minimize the cross partition edges between partitions. This reduces the chance for communities based on graph structural information to be spread across multiple partitions. While many community detection methods have been proposed [1] we focus in this paper on the recent Louvain community detection algorithm [10] which performs community detection by using a greedy modularity maximization approach [11]. The aim is to improve the performance of the algorithms's first iteration which takes on average 79% of the total algorithm time (cf. Section III).

We validate our approach by using an MPI implementation on a HPC cluster. To our knowledge this work is not only the first one to propose a distributed memory parallel extension to Louvain algorithm but also the first to take advantage of the graph partitioning to speed-up the community detection. The main contributions of this paper are the following:

1. We propose a distributed memory parallel algorithm extending the Louvain method by making its costly first iteration embarrassingly parallel without any noticeable loss in final modularity.
2. We show that by using different techniques for selecting the vertex traversal order of the Louvain method, we can further improve the performance.
3. We prove the efficiency of our algorithms by using a random community graphs ranging from 250K up to 16M vertices.

The rest of the paper is structured as follows: Section II presents some of the main results related to parallel algorithms for graph partitioning and Louvain community detection; Section III gives a brief overview on the Louvain method; Section IV outlines our proposed approach; Section V describes the

978-1-4799-6233-4/14/$31.00

experimental setup and discusses the obtained results; and Section VI outlines the main achievements of this work.

II. Related Work

Graph partitioning is a technique which aims to split the graph k-ways such that the edge cuts are minimized while trying to balance the number of vertices in each partition. Most variants of the problem are $\mathcal{NP}$-hard [1]. Kernighan et al. [12] proposed one of the earliest partitioning algorithms. The algorithm tries to optimize a benefit function defined as the difference between the number of edges inside partitions and the ones between them. Existing techniques used in graph processing tools such as Apache Giraph [8] rely on simple hashing or vertex ordering based partitioning. A wide array of multi-level partitioners are implemented in the METIS library [9]. A multi-level parallel partitioning algorithm has been presented in [13].

Recently Kirmani et al. [14] proposed a parallel graph partitioning algorithm based on a geometric scheme that offers good edge cuts. The algorithm is shown to scale better than parallel METIS.

Efficient parallel partitioning algorithms for large social networks have been proposed as well [15]. The algorithm relies on label propagation and on a parallel evolutionary algorithm to obtain quality partitions and is shown to produce better results then parallel METIS.

In this work we utilized parallel METIS partitioner (PMETIS) to perform the initial graph partitioning due to its wide availability and ease of use.

Community detection has been widely studied with Fortunato [1] providing a thorough overview of the main approaches. Most of the work however focused on sequential algorithms. Among the first algorithms to exhibit near linear complexity was presented [16]. The proposed algorithm is based on a label propagation approach. We focus next on existing work that addressed parallel algorithms for community detection with emphasis on parallel Louvain algorithm.

Parallelizing the community detection based on the modularity metric has received an increasingly attention in the past few years. Riedy et al. [17] proposed the first shared memory algorithm for dedicated architectures (Cray XMT and OpenMP). More recently other papers parallelizing the Louvain community method based on OpenMP have been proposed [18], [19]. While all these papers show good performance and modularity results they all take the same approach of using a shared memory architecture. We argue that in order to enable large scale distributed community detection we need to break free of this constraint.

III. Louvian Method For Community Detection

The Louvain method for community detection is a greedy modularity maximization approach [10]. Modularity is a widely-used metric for determining the strength of detected communities [11] and is defined as:

$$Q = \sum_{c \in C} \left[\frac{m_c}{M} - \frac{d_c^2}{4M^2} \right] \quad (1)$$

TABLE I: % Run time spent on first iteration of the Louvain algorithm for different graph sizes.

# Vertices	% Time for 1st Iteration
250,000	90.48
500,000	88.61
1,000,000	87.45
2,000,000	71.08
4,000,000	76.67
8,000,000	72.67
16,000,000	69.75

where C represents the set of all communities, M represent total number of edges in the graph, m_c represent total number of edges inside community c and d_c represent total degree of vertices in c. The modularity metric mainly tries to quantify how many internal edges are in the communities than the expected. For the normalized modularity value for a graph $Q \in [-1, 1]$ a value of 1 represents the ideal scenario.

Louvain method is an iterative algorithm. Each iteration consists of two major steps. It starts by initializing each vertex with its own community. In the first step, it chooses a vertex traversal order to scan through the vertices, which in a random order in the the original sequential implementation [1]. Then for each vertex in the graph it considers its neighbors communities and checks whether or not removing the current vertex from its current community and inserting it to neighboring communities results in any modularity gain. The vertex is added to the community which gives the maximum positive gain in modularity. Community remains unchanged if no positive modularity gain can be achieved. The process is repeated multiple times until a local maximum modularity is reached. In the second step a new graph is created by collapsing the detected communities into vertices. Intra-community edges are collapsed into a single self loop edge and the weight of this edge is calculated by summing up the edge weights of all intra community edges in the community. Multiple edges between each two communities are collapsed into a single edge by summing up the edge weight. The process continues by repeating steps one and two until no improvement in modularity between two consecutive iterations is observed.

Blondel et al. [10] suggested the Louvain methods runtime scales linear with the graph size and the first iteration of the algorithm is the most costly iteration in terms of computation time. We instrumented the sequential version of the algorithm to log the time spent in each iteration and observed that on average 79.53% of the time is spent there for most the community graphs. Table I shows the percentage of the time spent in the first iteration compared to the overall execution time. This result provided us with a clue on what part of the algorithm needs to be optimized. A synthetic community graph generator was used to generate the depicted graphs. More details on the generated graphs will be given in Section V.

IV. Proposed Approach

We modify the Louvain method and parallelize the first iteration in order to significantly reduce execution time (cf.

[1] https://sites.google.com/site/findcommunities/

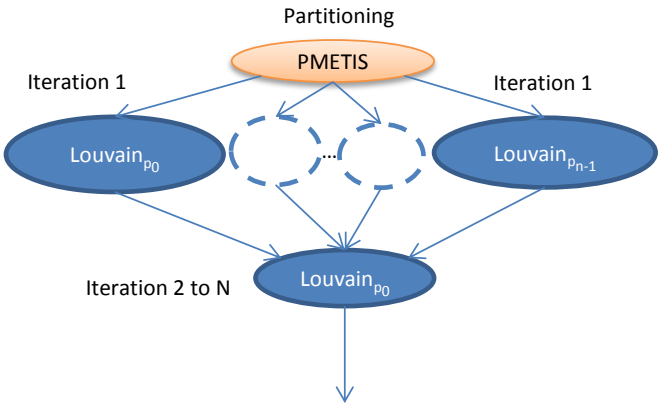


Fig. 1: Workflow of the proposed parallel Louvain algorithm.

Fig 1). For this we partition the graph using PMETIS. The parallel implementation was implemented a using GNU C++ message passing interface implementation where each MPI process picks up its partition based on the process id and runs the first iteration of the Louvain method within the process locally ignoring cross partition edges. Next, the graphs need to be merged together at the master node. As all individual partitions had local vertex numbering we need to renumber all vertices across partitions continuously. Thus one more step is required before sending the data to the master node. Through an *all gather* operation the total number of vertices in each process is sent to all other processes. Each process p_i will receive the total number of vertices in each partition $\{N_0, ., N_{i-1}, N_{i+1}, ..N_{P-1}\}$ where P is the total number of processes. It then renumbers its vertices such that the vertex numbers associated to its partition start from a value n_{start_i} computed based on the values of all processes p_j with $j < i$ as follows:

$$n_{start_i} = \sum_{0}^{i-1} N_j$$

After renumbering, each process sends its renumbered intermediate graphs to a master process which aggregates them into a single graph considering cross partition edges and computes the modularity value. The generated graph represents the first level in the hierarchical community. We must note that at this stage the size of the graph is reduced significantly. The algorithm then proceeds with the rest of the iterations by applying the sequential Louvain on this graph. The pseudocode for this process is shown in Algorithm 1.

As explained in Section III the original Louvain method uses a random vertex ordering at the start of each iteration to traverse the vertices. Because communities are created by a recursive exploration of the neighboring vertices we argue that a method in which the initial vertices are selected based on the edge degree may allow a faster convergence. To prove this we investigated three different vertex ordering strategies:

1 Ordering based on ascending order of edge degree.

2 Ordering based on descending order of edge degree.

3 High degree vertex neighbor order.

Algorithm 1 Pseudocode of the Parallel Extension for the Louvain Method Executed On Each MPI Process.

```
1: procedure PARALLEL LOUVAIN
2:     LOAD-PARTITION(process id)
3:     improvement ← LOUVAIN-ONE-LEVEL                ▷ Louvain iteration
4:     if ¬ improvement then
5:         EXIT
6:     end if
7:     OUTPUT-COMMUNITIES                              ▷ Output communities
8:     CREATE-NEW-GRAPH                ▷ Collapse communities into vertices
9:     RENUMBER-VERTICES      ▷ Renumber the vertices using all gather
   operation
10:    if process id = 0 then      ▷ If current process is the master process
11:        MERGE-GRAPHS
12:        improvement gets true
13:        while improvement do
14:            improvement ← LOUVAIN-ONE-LEVEL
15:            OUTPUT-COMMUNITIES
16:            CREATE-NEW-GRAPH
17:        end while
18:    else                          ▷ If current process is a worker process
       SEND-CURRENT-GRAPH                ▷ Send graph partition to master
19:    end if
20: end procedure
```

While the first two strategies are self explanatory the ordering of vertices for the third case is determined as follows:

1) Scan through vertices in the descending order of the edge degree.
2) For each vertex add all not visited neighbors to a queue and mark them as visited (neighbors of a given vertex are visited according to the ascending order of the vertex ids).
3) The vertex order of the queue is considered as the traversal order.

V. Experimental Results

A. Environment

The performance of our approach versus the sequential implementation of Louvain algorithm was evaluated by running a series of experiments on the University of Southern California's High Performance Computing Center (HPCC) cluster[2]. The HPCC cluster consists of heterogeneous computing nodes. All benchmarks were executed as single batch jobs of 16 compute nodes with 8 cores per node to address this issue. Experiments were conducted multiple times and the presented results are consistent with the results of all other rounds of experiments. Each node consists of two Quad-core AMD Opteron 2376 2.3 GHz processors.

B. Data Sets

Synthetic graph datasets were used in the experiments. The reason for choosing synthetic graphs is to have control over graph sizes to conduct scalability experiments over varying graph sizes. We used Fortunato's benchmarking suite for community detection [20] and generated 7 graphs (cf. Table II). This benchmarking package contains elaborate graph generation algorithms for undirected and unweighted graphs with community structures. In addition, the graph generation process allows fine control over many properties of the

[2]http://hpcc.usc.edu/

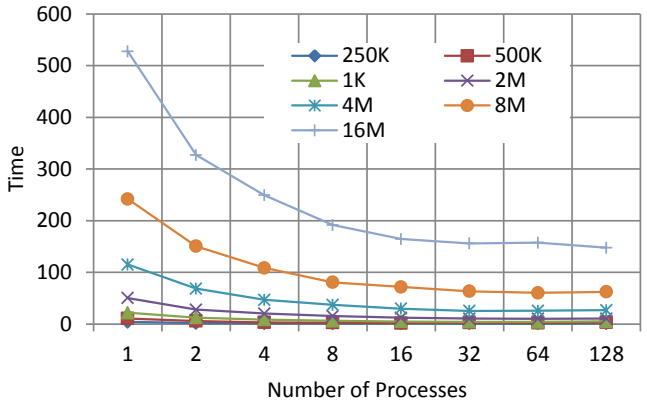


Fig. 2: Execution time when scaling up number of processors for different graph sizes.

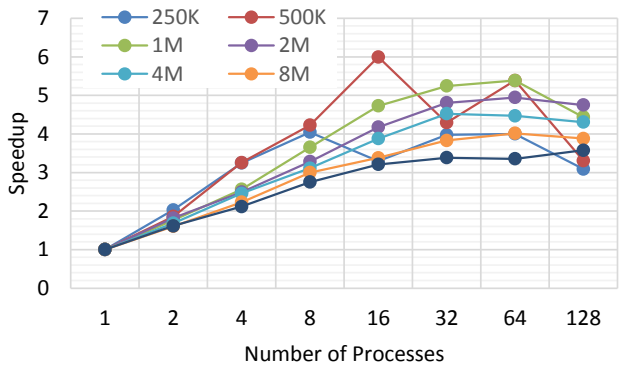


Fig. 3: Speedups compared to the sequential implementation when scaling up number of processors for different graph sizes.

produced synthetic graphs, including average and maximum degree distribution, and community overlap. But production of the largest graph in our dataset, e.g., 16M nodes, required a compute node with 64GB of memory at HPCC. That constrained us from generating larger graphs. A brief summery of data set properties are shown in Table II.

TABLE II: Data sets statistics summary.

# Vertices	#Edges
250,000	936,948
500,000	1,874,686
1,000,000	3,751,912
2,000,000	7,503,410
4,000,000	15,009,838
8,000,000	30,024,411
16,000,000	60,067,971

C. Results

We compared our parallel algorithm with the original Louvain algorithm implementation and also tried to compare our results with the shared memory implementation presented in [18]. However we found that the version used by authors of shared memory implementation does not scale beyond graphs larger than 10,000 vertices.

The main focus of this experiments was to analyze the scalability of our approach. We conducted series of experiments scaling the problem in two dimensions, namely with increasing number of processors and increasing graph sizes.

Figure 2 shows the change in total execution time when scaling up the number of processors for parallel implementation. We can see our approach outperforms the sequential version for all configurations. We also observed that the number of iterations of our algorithm remained the same as the sequential Louvain method. This is an indication that the partitioning of the graph for the first iteration did not split the communities in the first level(or had a very minimal impact). We notice however that the performance improvement decreases when scaling up number of processors for all graph sizes. The same behavior can be observed in Figure 3 where speedups start to

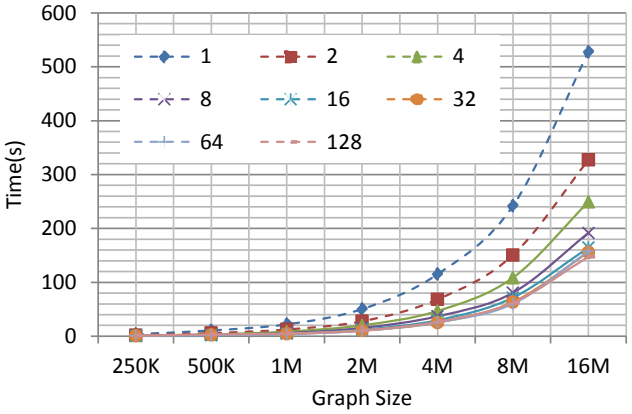


Fig. 4: The execution time when scaling up graph sizes with different number of processors.

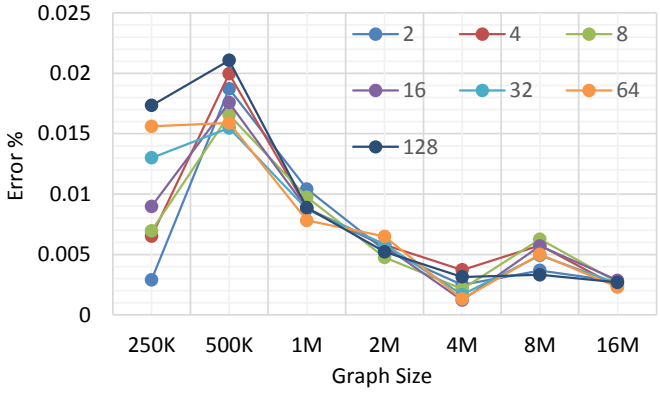


Fig. 5: Change in the percentage error of final modularity with graph sizes.

flatten for most of graphs after scaling up to 16 - 32 processors. The reason behind this is due to the upper bound enforced from the Amdahl's law [21] and implementation overheads.

As shown in Figure 4 we can see the gap between the runtime of both sequential and parallel implementations increases with graph size. This is an indication of good

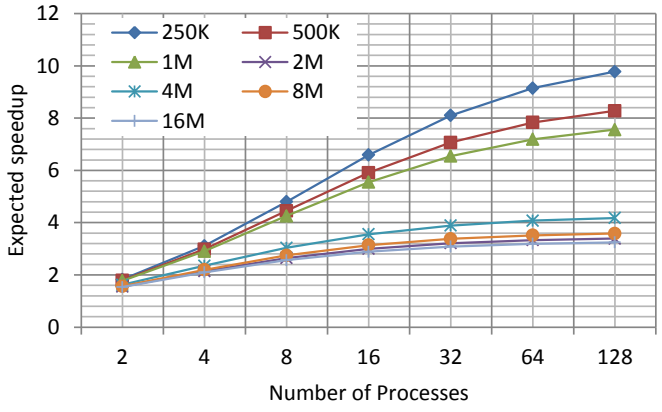


Fig. 6: Expected (theoretical) speedup for the parallel implementation when scaling up the number of processors for different graph sizes.

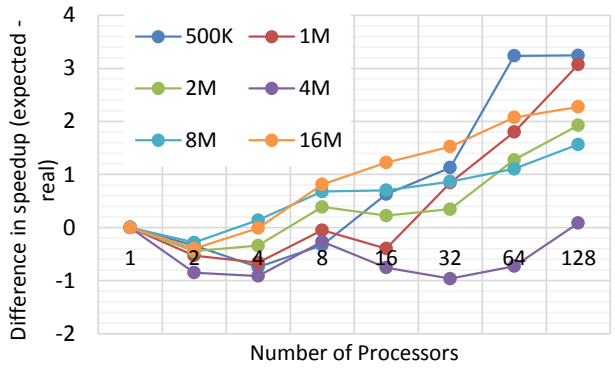


Fig. 7: Difference between expected speedup and actual speedup for parallel implementation when scaling up number of processors for various graph sizes.

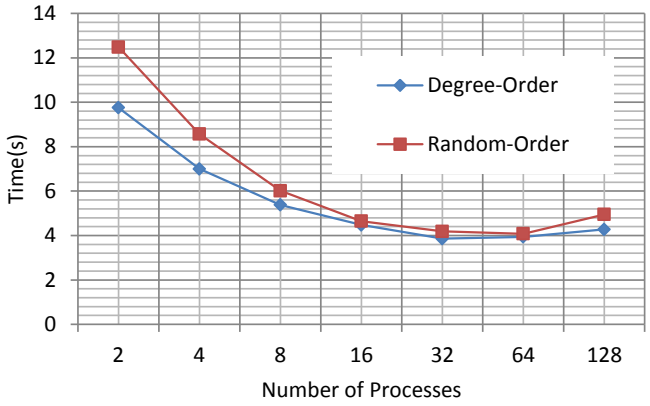


Fig. 8: Runtime behavior when using vertex ordering based on descending order of edge degree compared to random vertex order. Graph size: 1m vertices

scalability of our approach for larger graphs. But we can observe the improvement in runtime reduces when scaling up the processors up to 16-32 due to the same reasons explained before.

The correctness of the community detection algorithms is of a main concern when trying to gain parallel speedups. As an example if we ignore cross partition edges in community detection algorithms it might cost us in terms of final modularity due to incompleteness in detecting communities that span over multiple graph partitions. This poses the need for a fine balance between performance of algorithm and correctness of results. Figure 5 shows the behavior of the percentage error calculated using the difference in modularity between the sequential and parallel algorithms. Error E is calculated based on relation 2.

$$E = abs(\frac{q_p - q_s}{q_s}) \quad (2)$$

Where q_s and q_p represents the final modularity got from sequential algorithm and our approach respectively. Results show that our approach finds communities with almost same quality as the sequential Louvain method (less than 0.025% error). One notable observation is that the percentage error reduces with the graph size. This is expected since the probability of partitioning communities into multiple graph partitions reduces for larger graphs since PMETIS tries to reduce the number cross partition edges between partitions. But for smaller graphs since PMETIS is constrained by number of partitions it will have to partition the graph into given number of partitions irrespective of increase in number of cross partition edges. This increases the chance of communities to be split across partitions during the partitioning process. Since our method ignore the cross partition edges in the first iteration the final community quality will be significantly effected for very small graphs.

As explained in Section IV our approach parallelize the first iteration of the Louvain method. Amdahl's law bounds our speedups based on the percentage execution time of this first iteration compared with the total execution time. To model the expected speedups for our approach timing results were collected for each iteration of the sequential implementation for different graph sizes. Figure 6 shows the predicted speedups for the parallel implementation and Figure 7 plots the change in difference between expected and actual speedups. Expected speedups were calculated in an optimistic manner ignoring the communication times and assuming linear run times with the graph size [10]. We can see that for all graph sizes the difference becomes larger with the graph size. This is expected due to the increase in overheads, but interestingly we see super linear speedups for almost all the graphs with initial small number of MPI processors. This is mainly due to the partitioning process where after partitioning the convergence rate (number of iterations required in the first step of Louvain to reach the local maximum modularity) of Louvain method iteration within the partition becomes higher than the expected. This is mainly due to the reduction in search space of the first step of Louvain method since we ignore cross partition edges in the first iteration.

We also evaluated the initial three vertex ordering strategies on the same graph data sets (cf. Section IV). Strategy 1 and 3 affected the overall run time in a negative way. Convergence rate of the algorithm reduced dramatically when first strategy was used. The computation complexity of the 3rd strategy

outperformed the improvement in convergence rage. As shown in Figure 8 we were able to get some marginal performance improvement when using the second strategy. But the improvement in convergence rate reduced when we increase the number of graph partitions.

We also conducted some preliminary experiments on a real world social network [3] with $\approx$ 16 million vertices and 30 million edges. Initial results indicated the same behavior with our approach, i.e., $\approx$ 2.6 speed up with 2 processors while final modularity of serial and parallel versions being 0.7162 and 0.7051 respectively.

VI. Conclusions And Future Work

Community detection in large graphs is receiving increasingly traction due to the unprecedented growth in internet and online social networks. Our work combines an existing graph partitioning technique which minimizes the cross partition edges, with the Louvain community detection method. We specifically target the costly first iteration of the sequential algorithm. Results showed our approach scales for large graphs giving significant performance improvements. We show that processing graph partitions independently in the first iteration ignoring cross partition edges does not impact to the quality of the final result.

Our evaluation on different vertex ordering strategies suggested that ordering the vertices in the descending order of edge degree can further improve the convergence rate of the Louvain method while giving the same final modularity.

We plan to further evaluate our approach along few directions. First, the final modularity of our method is highly dependent on the used graph partitioner. We plan to evaluate our method on few different graph partitioners and draw some conclusions that can help users decide on the best partitioning method they should use for Louvain. Second, we will explore strategies to improve the modularity in case there are communities that span across graph partitions by performing some communication between graph partitions.

In this work we did not perform extensive experiments on real world graphs. Even though preliminary results on real world graphs shows our approach holds, we would like to evaluate this approach on different classes of real world graphs with known ground truth communities [22].

Acknowledgment

This work was supported by a research grant from the DARPA XDATA grant no. FA8750-12-2-0319. Authors would like to thank Prof. Cauligi Raghavendra of University of Southern California, Jonathan Larson of Sotera Defence and Alok Kumbhare for their feedback.

References

[1] S. Fortunato, "Community detection in graphs," 2010, uRL: http://arxiv.org/abs/0906.0612 (accessed May 13, 2014).

[2] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 85–98.

[3] http://snap.stanford.edu/data/soc-pokec.html

[3] S. Asur, S. Parthasarathy, and D. Ucar, "An event-based framework for characterizing the evolutionary behavior of interaction graphs," *ACM Trans. Knowl. Discov. Data*, vol. 3, no. 4, pp. 16:1–16:36, Dec. 2009.

[4] twitter blog, "New tweets per second record, and how," 2013, uRL: https://blog.twitter.com/2013/new-tweets-per-second-record-and-how (accessed July 19, 2014).

[5] A. Pothen, "Graph partitioning algorithms with applications to scientific computing," in *Parallel Numerical Algorithms*. Springer, 1997, pp. 323–368.

[6] Y. Simmhan, A. G. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. S. Raghavendra, and V. K. Prasanna, "Goffish: A sub-graph centric framework for large-scale graph analytics," in *Proceedings of the 20th International Conference on Parallel Processing*, ser. EuroPar '14, 2014, p. accepted.

[7] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM. New York, NY, USA: ACM, 2013, pp. 22:1–22:12.

[8] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," in *Hadoop Summit*, 2011.

[9] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.

[10] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, no. 10, p. P10008 (12pp), 2008.

[11] M. E. J. Newman, "Modularity and community structure in networks," *Proceedings of the National Academy of Sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[12] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[13] G. Karypis and V. Kumar, "Parallel multilevel graph partitioning," in *Proceedings of the 10th International Parallel Processing Symposium*, ser. IPPS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 314–319.

[14] S. Kirmani and P. Raghavan, "Scalable parallel graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 51:1–51:10.

[15] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," *CoRR*, vol. abs/1404.4797, 2014.

[16] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, 2007.

[17] J. Riedy, D. A. Bader, and H. Meyerhenke, "Scalable multi-threaded community detection in social networks," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, pp. 1619–1628.

[18] S. Bhowmick and S. Srinivasan, "A template for parallelizing the louvain method for modularity maximization," in *Dynamics On and Of Complex Networks, Volume 2*, ser. Modeling and Simulation in Science, Engineering and Technology. Springer New York, 2013, pp. 111–124.

[19] C. Staudt and H. Meyerhenke, "Engineering high-performance community detection heuristics for massive graphs," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013, pp. 180–189.

[20] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 35, pp. 75 – 174, 2010.

[21] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[22] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, ser. MDS '12. New York, NY, USA: ACM, 2012, pp. 3:1–3:8.

Enabling Real-time Pro-active Analytics on Streaming Graphs

Charith Wickramaarachchi, Marc Frincu,Viktor Prasanna
University of Southern California
Los Angeles CA 90089 USA
{cwickram,frincu,prasanna}@usc.edu

***Abstract*—Massive growth in areas like social networks and cyber physical systems pushes the need for online analytics of high velocity graph structured data streams. Its intrinsic velocity and volume challenges existing graph processing frameworks like Google Pregel and traditional stream processing systems. Consequently we need a shift in the way these systems are design and design new data and execution modes, and architectures. In this paper we propose and position a new analytical model for processing high velocity graph structured streaming data while providing an architecture and discussing some research challenges that need to be addressed before it can be accomplished. A couple of motivating use cases are mapped to the model and architecture are also discussed.**

***Keywords*-graph streams;graph analytics; time series; elastic architectures; data models;**

I. Introduction

The term Big Data was first referenced in 1997 by Cox and Ellsworth [1] and was used to refer to large data sets that do not fit the main memory or local disk. Even before that physical sensors or monitoring software were employed to gather data from various sources ranging from astronomical surveys and DNA sequencing to traffic monitoring and social networking. Gartner [2] defines Big Data as data having large volume, variety and velocity, although some have added other *v*s such as veracity to it. Due to the nature of the gathered datasets research in the past decades was more focused on the volume aspect. The current rate at which information from social media or cyber-physical is gathered brings velocity to the table. Many applications dealing with this kind of streaming data require real-time processing, e.g., threat prevention and real-time optimization. High velocity data processing systems Twitter Storm [3] and Amazone Kinesis [4] were introduced as a result of this.

Much of this Big Data streams exhibit structural patterns which can be naturally modeled as graphs – e.g., social media like Twitter or Facebook; road network traffic; smart grid power grid and customer network. Current systems, be they vertex (e.g., Pregel [5]) or subgraph (e.g., GoFFish [6]) centric are designed to deal with volume and variety. Currently however their is an increased need to process these high velocity data streams in real-time based on online model which notifies applications on occuring complex events within the graph structure. These can help users and applications in optimization decisions by investigating possible causality patterns between the detected patterns, and by taking pro-active decisions on possible upcomming events – e.g., social network evolution based information coming from the Semantic Web and social networks; road network monitoring and traffic optimization; power management optimization in smart grids. The real-time constraint combined with a varying input rate and volume for the data streams brings to the table the need to elastically scale the execution/storage platform so that costs are minimized and the time constraint is satisfied. To date no suitable models for streaming graph processing in distributed elastic environments exist.

Real-time streaming graph processing requires an asynchronous push-based model designed for scalable architectures. In this way users: (1) do not have to explicitly check for events at specific time stamps, instead they use a continuous query language based on a publish-subscribe model; (2) the real-time constraint of minimizing the lateness of each event detection is achieved through graph aware online scaling; and (3) analytics on each job can run asynchronously with respect to the other jobs.

In this position paper we take a look at the main issues that arise in designing a platform for pro-active analysis of streaming graphs. In our context these graphs are defined as time-evolving graphs with a fast update rate of their vertices, edges and associated attributes. To better understand the challenges we first look at a possible model for this class of graphs and sketch out a platform architecture (cf. Sect. III). As seen in Sect. II no current system offers the full model and software stack to deal with these kind of graphs. While we make no assumption on the underlying platform the elastic nature of the applications makes it ideal for modern scalable infrastructures such as clouds. The model and architecture are mapped on a series of use-cases in order to prove their validity.

Our main contributions can be summarized as follows:

- we propose and discuss a novel pro-active push-based analytic model for processing streaming graphs;
- we present a distributed scalable architecture that will enable such a model and discuss its main components;
- we propose a continuous query language called CGQL (Continuous Graph Query Language) for pro-active graph stream processing that enables users to perform real-time graph analytics without having to deal with the underlying graph structure or distributed system;

- finally, we identify a series of research challenges that arise from the above and related to: scalability, data partitioning, memory representation and storage, execution model, and translation between abstract CGQL and concrete graph queries.

The remaining of the paper is structured as follows: Section II gives an overview of existing related work and its limitations for real-time streaming graph analytics; Sects. III and IV deal with the proposed model and architecture; Sect. V discusses CGQL; Sect. VI presents several use-cases based on the proposed model and maps them on the envisioned architecture; Sect. VII depicts the challenges set out by the future research directions; and finally Sect. VIII summarizes the main key-points of this research.

II. Related Work

Graph analytics and mining gained a lot of traction in both research and industry in the last few years due to rapid development in areas like social/communication networks and computational biology. Even though lot of work has been done on graph processing and management applying them in emerging domains such as real-time Big Data analytics becomes a challenge due to the volume, velocity, variety and veracity of the data [7].

Large scale graph mining. Processing of large scale graphs is a well studied problem. Both parallel and distributed algorithms have been developed for large scale graph processing. Due to the massive expansion of the web and social networks, large scale graph mining regained its momentum in last few years. With the introduction of Map Reduce framework [8] efforts have been made to come up with large scale graph processing frameworks which runs on commodity clusters and clouds, e.g., Google Pregel [5], Trinity [9], Pegasus [10], GraphLab [11]. All of these frameworks focus on large scale static graphs and not streaming graphs.

Time evolving graphs. Some graphs like social/communication networks exhibit evolving characteristics over time. Managing and processing these dynamic graph-structured data started to get some attention in the research community [12] [13] [14] [15] [16] [17] [18] [19] [20]. Time evolving graph processing can be categorized into two ways: processing of graph snapshots, and processing of graph updates. In [12] [13] snapshots of time evolving graphs are created. Incremental algorithms and parallel algorithms and approaches mentioned in [14] [15] [16] [17] [18] [19] [20].

Our approach is to combine both in different phases of the process. For in-memory fast event detection in time evolving graphs we use incremental and distributed algorithms which process the unprocessed updates to the graph. We also create and persist temporal snapshots for offline batch processing of time evolving graph data.

None of the existing techniques work on fast complex event detection in large scale time evolving graphs and they do not consider attribute changes of the graph components (vertices and edges) to do the analytics. We believe these to be important factors when it comes to social/interaction graph analytics. Another major difference is the targeted platform. The algorithms and methods proposed in [14] [15] [16] [17] [18] [19] [20] rely on high performance shared memory machines which can store the graph in-memory. We target cloud platforms due to their inherit elasticity. As a result, our work involves the development of distributed algorithms and models, some of which being possible extensions (that support attribute changes) of existing parallel algorithms [15] [18].

Dynamic graph re-partitioning. Graph partitioning is a well studied problem in parallel graph processing area. The main idea is to assign vertices/edges in a graph to different parallel graph processing workers to minimize the communication between workers [21].

After the introduction of iterative graph processing programming models like Google Pregel [5] and its successors the problem of dynamically re-partitioning the graph has been studied [22]. Current studies on this problem is only done on static graphs where to the best our knowledge no prior work has been done for doing dynamic re-partitioning of time evolving graphs.

Complex event processing. Data stream processing systems are getting traction due to increase popularity and usage of sensor systems, social media streams, etc. As a result, we notice new emerging systems for handling this high data velocity [4] [3]. Data stream processing systems focus on doing calculations on the data streams [23]. SQL like languages have beeen proposed and implemented to input complex event matching queries, e.g., SASE and RAPIDE-EPL [24]. To the best of our knowledge, no system has been proposed to enable complex event processing on time evolving graphs which consider graph updates as events. Here the challenge lies in managing and processing the large underlying graph state to detect events. Where in existing CEP systems this state is very small and can be fit into the main memory of a single machine.

Graph queries and query languages. The relational database model was the de facto method for storing enterprise application data in a decade back. But relational model and SQL do not have enough native support to efficiently represent and query graph structured data. With the increasing popularity of XML, social networks and the ontologies, graphical query languages and graph data management systems have become a need. Examples of proposed graph query languages include GraphQL [7] GraphLog [25] and GraphDB [26].

To our best knowledge all the existing graph query models are based on a data pull model where user executes a query one time to pull the matching results. We believe that due to the fast changing nature of the targeted graphs, there is a need for push based query model which will continuously push matching complex events to the users

once they mention the graph patterns they are interested in. Some work on defining and detecting events in time evolving graphs [12] has been done, but it does not provide a general query model for providing these event patterns.

Concluding, based on our analysis of related work, none of the existing solutions offers a suitable model – at data, execution and query levels – for enabling large scale pro-active real-time analytics of streaming graphs.

III. Proposed Model

Previous work [13] has defined streaming graphs as an unbounded sequence of graph instances taken at discrete time steps. We argue that this approach induces too much overhead in terms of storing the entire graph structure and creating efficient indexes to access the data. Instead we propose a novel approach in which a **graph stream** contains only the update information not the topology or attribute information. The topology with its semantics will be built at runtime by processing the events generated in the stream.

A. Streaming Graphs

Graphs with attributes or Attributed Relational Graphs (ARGs) as named in domains like image processing [27] can be used to model graph structured data similar to what we find in social and communication networks. These graphs can be seen as an extension to traditional graphs by enabling vertex and edge attributes.

Formally and ARG $G_A = (V, E, A_V, A_E)$, where V, E, A_V, A_E represent the set of vertices, edges, and vertex and edge attributes respectively. The vertex and edge attribute sets are defined as $A(V) = \{id, \alpha_1, ...\alpha_m\}$, and $A(E) = \{id, \beta_1, ...\beta_n\}$, where id is a time invariant unique identifier and α_k, β_k represent name value pairs.

Given an ARG we define a graph stream as an unbounded sequence of updates to G_A: $\mathcal{S} = \{\delta_1, ..., \delta_n, ...\}$, where an update δ_i can be any of the following:

- vertex addition (+v): $V = V \bigcup \{v\}$;
- vertex removal (-v): $V = V \setminus \{v\} \wedge A_V = A_V \setminus \{a_i : i.id = v.id \; \forall i \in V\}$;
- edge addition (+e): $E = E \bigcup \{e\}$;
- edge removal (-e): $E = E \setminus \{e\} \wedge E_V = E_V \setminus \{e_i : i.id = e.id \; \forall i \in E\}$;
- vertex attribute addition (+a_v): $A_V = A_V \bigcup \{a_v\}$;
- vertex attribute removal (-a_v): $A_V = A_V \setminus \{a_v\}$;
- edge attribute addition (+a_e): $A_E = A_E \bigcup \{a_e\}$;
- edge attribute removal (-a_e): $A_E = A_E \setminus \{a_e\}$.

Without loosing generality we consider an update on a graph to be instantaneous, i.e., δ_k is dimensionless. This allows us to define fine grained events that take place on graph streams which are used in graph-oriented analytics to determine application specific time evolving patterns.

In practice we extract these types of graph streams from online social media like Twitter [28] and Friendfeed [29].

We define a Streaming ARG (SARG) as an ARG that evolves over time under the effect of $\mathcal{S}$: $G_A^{(t)} = f(G_A, \mathcal{S})$.

B. Elastic Buffer

The high velocity nature of graph streams and the real-time processing constraint requires us to develop optimized data structures for storing only the most relevant information for online processing. For this we define elastic streaming graph buffers, $\mathcal{B}^{t_s \to t_e} = \{\delta^{(t_s)}, ...\delta^{(t_s+1)}, ..., \delta^{(t_e-1)}\}$, where the length of the buffer is variable and equal to $t_e - t$. These buffers contain the latest most relevant stream data. Due to memory and performance issues these buffers cannot extend indefinitely in the past, therefore we define periodic checkpoints based on the stream rate and volume and not only on time intervals. This allows us to define both a spatial and a temporal based snap-shooting policy: $\mathcal{L}^{t_i \to t_{i+1}} = f(\lambda^{t_i \to t_{i+1}}, DV^{t_i \to t_{i+1}})$, where $\lambda^{t_i \to t_{i+1}}$ represents the input rate, $DV^{t_i \to t_{i+1}}$ is the data volume, and $\mathcal{L}^{t_i \to t_{i+1}}$ is the buffer length.

Given a checkpoint the buffer contains only information from that particular checkpoint onward. The buffer is the centerpiece of our model as we can use it to perform Iterative Asynchronous Bulk Parallel (ItABP) graph analytics, online graph repartitioning and resource scaling to efficiently process these analytics in real-time.

C. Graph Instances

A graph instance G_t is a checkpoint performed on a graph at a given time. The set of instances produces a discrete time series $G_{t_\infty} = \{G^{(1)}, ..., G^t, ...\}$ taken over t. A transition from one instance to another – i.e., the difference $G_{t_i} - G_{t_{i+1}}$ – can be represented by the SARG function applied inside the interval $[t_i, t_{i+1}]$: $f(G^{t_i}, \mathcal{S}^{t_i \to t_{i+1}})$. Figure 1 shows an example of three consecutive instances and the buffers associated with each transition.

D. Towards Fast Pro-active Analytics on Streaming Graph

We propose a novel pro-active graph analytics model which processes graph streams and continuously pushes results to the user in form of events named Complex Graph Events (CGE). We differentiate CGE from low level graph events stored in the buffer. As shown in Fig. 2 the platform processes the buffer based on the stream and user query and emits CGEs. As an example we assume an input stream of social network feeds and a user query for determining the evolution of a given user's friend social network. the system At a platform level the query is translated into graph specific language where the user's social network is equivalent to a connected components, and whether or not two users are connected through a path is determined through a reachability change between their corresponding vertices. The platform will monitor the evolution of the reachability between vertices and emit corresponding events. These low level CGEs will be mapped back to the query

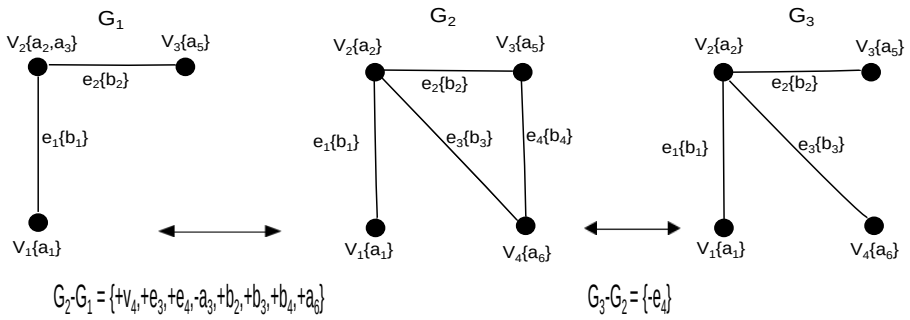


Figure 1. Example of three consecutive graph instances.

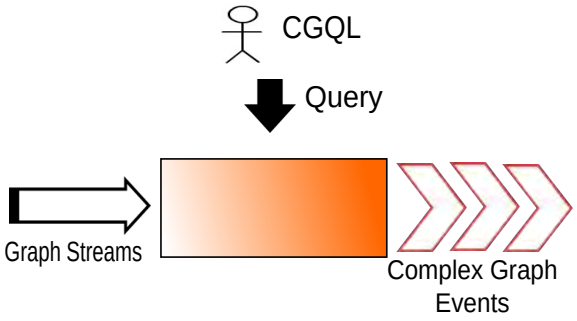


Figure 2. High level conceptual overview.

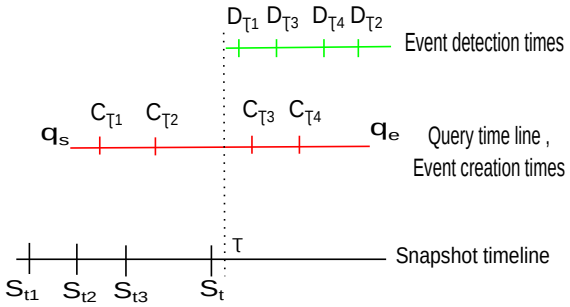


Figure 3. Graph snapshot creation and event detection.

semantics and the user will see in real-time how the targeted social network evolves.

As mentioned we rely on the buffer to handle the input stream for fast processing. The buffer stores only those updates arriving after the last graph instance snapshot. In Fig. 3 S_t is the latest snapshot taken at time t. In a normal scenario a user will input a query to continuously monitor an event within either a specified time frame $[q_s\ ,\ q_e]$ or continuously ($[q_s\ ,\ \infty)$) until a cancellation query is dispatched – note that $q_s < t$ is possible. Based on the buffer data, the user query, and possibly on one or more snapshots if the time frame starts in the past, the platform will emit CGEs in real-time. Four cases are possible:

- $q_s \geq t$: the platform processes both the buffer and the information present in the snapshot G^t or a query specific index created at time t;
- $q_e \leq t$: the platform processes only the information stored in the graph instances taken in the specified time frame. This is the usual case of batch processing frameworks like Pregel [5] or GoFFish [6];
- $q_s < t$ and $q_e > t$: this case combines both real-time and batch analytics. Two processes are created for each: one job will detect complex events for the duration of $[q_s, t)$ by relying on graph instances, and the other will continuously detect events in the $[t, q_e)$ time frame.

Given a query time τ the real-time constraint of our model requires that the lateness of our detection events is minimized:

$$\begin{cases} \min \sum_{\tau_i} (D_{\tau_i} - C_{\tau_i}) & \forall \tau_i > \tau \\ D_{\tau_i} < q_e & \forall \tau_i \leq \tau \end{cases} \quad (1)$$

where D_{τ_i} represents the time when event τ_i is detected and C_i is the actual time the event was triggered. Ideally $\sum_{\tau_i} (D_{\tau_i} - C_{\tau_i}) \rightarrow 0\ \forall \tau_i > \tau$.

IV. Proposed Architecture

To better understand the challenges posed by enabling pro-active streaming graph analytics we present in this section a possible scalable software architecture to address the problem. Figure 4 gives an overview of its modular design. We only provide the high level details of the system. Modeling and algorithmic details related to dynamic repartitioning and the ItABP are out of the scope of this paper.

The architecture is based on a publish subscribe mode in which users subscribe to specific CGEs by issuing Complex Graph Queries (CGQs) (cf. Sect. V). These are processed by the different modules as depicted next and the results are published and placed under corresponding topics for users to access based on their subscription. In this way we enable pro-active streaming graph analytics so that users do

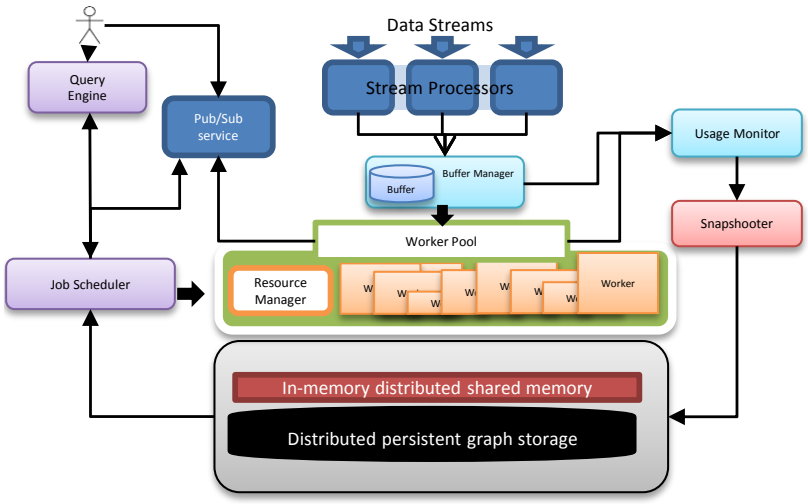


Figure 4. Proposed system architecture.

not have to explicitly ask for CGEs but instead they are automatically notified of their occurance.

The communication between the various modules is based on message queueing systems such as RabbitMQ [1].

A. Stream Processors

The system accepts multiple data streams which contain graph event information. Stream processors process these data streams and emit low level graph events. The processing logic can be specific to the stream that is being processed, hence the user is allowed to implement custom transformation logic to transform and extract the graph events from the data stream. Stream processors can act as a first level of processing for time series data streams but this level of analytics is out of the scope of this research. The generated low level graph events are then directed to the Buffer Manager where they are buffered in the in-memory event buffer (cf. Sect. III-B).

B. Buffer Manager

The Buffer Manager is responsible for storing the current graph event buffer and distribute it to the corresponding workers based on data and query locality. It is also communicates to the Usage Monitor information about the stream rate and volume. The Usage Monitor later aggregates the data and sends it to the Graph Instance Snapshooter which performs dynamic repartitioning for the next graph instance, and also to elastically scales the Workers.

C. Graph Instance Snapshooter

The Snapshooter is responsible for creating graph instance snapshots continuously by updating the most recent graph snapshot with the graph events in the buffer. The rate at which these snapshots are created as well as the repartitioning process are based on information provided from the Usage Monitor. The main reason for repartitioning is to improve the overall processing latency of the Workers. Two kinds of knowledge is needed to perform an efficient partitioning. First we require stream related information such as: rate and volume; second we rely on Worker specific information such as computation to communication ratio of workers; and communications patterns between existing Workers. It must be noted that the latter data is query dependent. On one hand a centrality query will only affect clustered data which is highly likely to be stored on a single Worker in order to reduce communication costs. On the other hand a reachability query and even a spanning tree is most likely to affect a large number of Workers and induce large communication overhead. The way the partitioning is performed affects the scalability of the Workers. While the specific partitioning and scalability algorithms are beyond our scope we will provide some high level insight when discussing the Worker pool.

After the new instance is created the previous one is moved to the persistent storage. Each time a new instance is captured it is partitioned based on historical information from the elastic buffer and statistics from the Workers so that that communication between workers is minimized. Workers are redistributed by elastically scaling them according to the new partitioning. The partitioning algorithm itself is beyond the scope of this article.

D. Query Engine

The Query Engine is handling the submitted CGQs. It acts as the interface between the Job Scheduler and the user. The CGQL is translated into an object model and submitted to the Job Scheduler. In doing so the engine creates a topic for the CGQ in the publish subscribe service and returns it

[1] http://www.rabbitmq.com/

to the user. Users can then subscribe to a topic to receive CGEs produced as a results of processing the query.

E. Job Scheduler

The Job Scheduler is responsible for scheduling CGQ jobs on the Workers. It takes a CGQ as input from the Query Engine and sends it to each Worker based on the current graph partitioning. As an example, the next reachability detection query:*Detect Reach(source:*$\{interests = foo\}$ *, sink:*$\{interests = bar\}$ *)*, outlines the user's interest in how groups with two interests are linked and how the links evolve over time though. Will be sent to each Worker which based on a semantic matchmaking will perform an index based search on all vertices with an attribute interests=*foo* linked through a path with vertices with attribute interests=*bar*. The path in our case can be set of edges where an edge represent a follow tweet action, a like event or a comment in a blog.

F. Worker pool

The Worker Pool consists of a Resource Manager and a set of Workers, i.e., self-contained distributed jobs which cooperate towards performing complex real-time graph analytics. Each worker is assigned a partition on which it performs the analytics. It will have access to the buffer feeding its subgraph with events, the latest instance snapshot and to the user query. To achieve full efficiency workers are able to scale either vertically (by allocating more resources for the virtual machine they run on) or horizontally (replicate themselves on another virtual machine) in case the Usage Monitor triggers a throughput alert. The actual resource allocation and worker execution is performed by the Resource Manager. Based on information retrieved from the Usage Monitor, the Resource Manager handles virtual machine and job failures, job reallocation and straggler mitigation.

Workers which detect CGEs will publish (push) that information to the corresponding topic so that the user wil be automatically notified on its occurance and have access to the related information, e.g., reachability status, cluster diameter, etc. They also report their usage statistics like processor/memory usage, communication statistics (e.g., which workers it communicates with and how frequent) to the Usage Monitor which will use them as parameters when determining the partitioning of the next graph instance.

G. Distributed Storage

The Distributed Storage consists of two layers: an in-memory distributed shared memory and a distributed persistent graph. The former is used to store the current graph instance and related indexes, while the latter contains all previous graph instances. For quick access the graph components are indexed using a distributed indexing scheme. Also some indexes are created based on most frequent user queries.

As explained in Sect. III the user query can span a time range, which can start before the current graph snapshot. Older graph snapshots stored in the persistent storage are used for detecting CGEs from a time before the current graph instance.

V. Continuous Query Language

To enable CGE detection users need a language with a wide degree of freedom in defining what they look for, and also a mechanism that frees them from continuously pushing for detecting new events. As seen in Sect. II query languages have become familiar in doing graph analytics and also time series mining. One solution may be a Continuous Graph Query Language (CGQL) which allows users to input a one time query and wait for CGE notifications as they are triggered by the graph analytics requested through the query. The CGQ will end as soon as the user will issue a canceling query or the time frame specified in the query is reached. A publish-subscribe approach is followed to enable this kind of continuous push-based model. To our knowledge, no prior work has been done on designing a continuous query language for graphs.

A CGQ needs to allow a wide range of operations required for performing analysis on streaming graphs:

DETECT *pattern[...]* ***[FROM*** *start_time* ***TO*** *(end_time]* | ***FOR*** *duration* | ***SINCE*** *duration]* ***[WHERE*** *optional_constrains]* ***[AS*** *topic]*

Next we discuss each of its components:

A. Query pattern

Query patterns represent graph patterns or analytics the user is interested in. Since there can be many application specific analytic that can be done on the graph structured data users can plug new patterns to the system. Each custom pattern will have its own query syntax and will extend the default Pattern interface. A new pattern definition includes CGE types and complex event schema for the respective outputs. An example pattern is explained in the Sect. V-E. When the user executes the query the system will return a topic name to which users can subscribe in order to receive CGEs which are detected and published by the Workers.

B. Query Duration

Optionally the query duration can be provided. By default the life span of the query is the infinite and starts at the moment it is submitted. The duration is defined using one of following constructs:

- *FROM start_time TO end_time*: defines the time span of the query using bounds provided as Unix cron expressions[2] that denote the start and end time of the query;

[2]http://www.nncron.ru/help/EN/working/cron-format.htm

- *FOR duration*: defines the time span of the query using the query duration. The query will start executing immediately and run until the duration is reached;
- *SINCE duration*: queries for CGEs that took place in the past.

C. Optional Constrains

A couple of optional constrains allow to further limits on the query, e.g., edge label constrains for reachability queries or generic constrains like a request for a time ordered delivery of CGEs. A constrain can also contain other queries. Finally it can contain keywords like PREDATES or SUCCEEDS to indicate its temporal relationship to other queries. Section VI will present a detailed example.

D. Extending the Continuous Query Language

The CGQL can be carried as a part of full time series data processing language. Users can optionally specify the output stream of the CGEs as a part of CGQ which can then be used as inputs to an event stream part of a complex event processing query. This allows users to perform broader analytics beyond CGE detection. An concrete example will be provided in Sect. VI.

E. Example Query

Next we provide a CGQL example based on reachability query patterns. These can detect two types of CGEs: (1) new reachable source sink node pairs (reachability pattern); and (2) reachable source sink pairs that will become unreachable (unreachability pattern).

At output the CGE contain the following information: time stamp of the event; status (reachable or unreachable); status vertex (ID and attributes); sink vertex (ID and attributes); and edge label constrains.

DETECT Reach(source:$\{name = Bob, age = 33\}$ *, sink:*$\{age < 30,\ lives\_in = London\}$*) FOR 20h WHERE edge:*$\{type = follow\}$

This query will continuously detect streaming graph CGEs for 20h. The pattern it looks for is the reachability between source nodes with attribute *name=Bob* and *age=30*, and sink nodes with age less than 30 and lives_in attribute with value 'London'. reachabilities with edges having the attribute type value 'follow' will be considered.

VI. Motivating Use Cases

In this section we present two use cases from distinct fields that fit our streaming graph model: a smart grid and a computer network use case. Their choice aims at showing the generality of our model for streaming graph analytics and to prove the effectiveness of our model and architecture by mapping the applications on them.

Smart Grid. Due to increasing demand for efficient electrical power management and conservation more and more power grids are updated as smart grids. As a result smart meters installed at the consumer premise communicate in real-time with the power utility. Normally, smart meters send out power usage information to the utility in a synchronous manner at a predetermined time interval. This allows utilities to forecast and predict incoming power demands and perform demand response activities in real time. Demand response is done by either voluntary curtailment where utility sends a curtailment request to a selected subset of identified customers or direct load control through smart appliances. Either way the utility should identify and predict peak demand zones and determine based on these the subset of customers to be select in the demand response.

Smart grids can be modeled as graphs where consumers are represented as vertices and power connections as edges. This is a structurally static graph where only the vertex and edge attributes – for power consumption – change over time as a result of the fluctuating consumption. On top of this network we can overlay a the customer social network where they interact via tweets, Facebook, etc. The mapping of the social network users and the smart grid customers is beyond our scope but we can assume the system to be integrated in the smart grid portal to facilitate the log in and communication with the utility. The social network produces a fast changing SARG and a slower topological change. Figure 5 exemplifies this type of graph with two types of edges – for the social network and for the smart grid topology – and two types of vertices (for the customers and for the smart grid relay and distribution stations).

Wide Area Networks (WAN). A large scale service provider like Google has a geographically spread WAN. For management and security reasons this WAN is in general composed of multiple networks. Service nodes which provide services connect to this WAN. This creates a two layered graph with one layer being the static network topology graph and the other the service layer with a changing topology based on the routing protocols. Depending on their attribute values vertices can be either physical machines or applications running on them, while edges are either physical routes or communication routes defined by the routing protocols. Figure 7 exemplifies this scenario.

A. Graph Pattern Detection

Areas with high demand or those which exhibit coalescent consumption patters during a day or within a demand response event are of particular interest to utility providers. The emergence of these clusters; their growth beyond some given threshold; their merging or even disappearance is of particular interest especially if they follow a certain temporal pattern or are connected with the social interaction between customers. Taking in consideration a possible causality between the social interaction and power consumption a CGQ could search for the formation of cluster consumption patterns preceded by several reachability events in the social

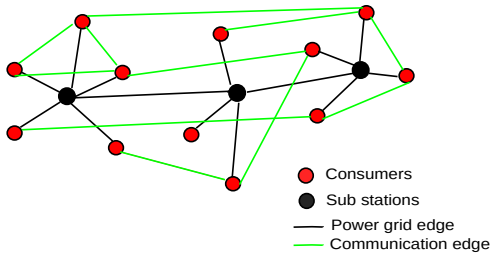


Figure 5. SARG for the smart grid use case.

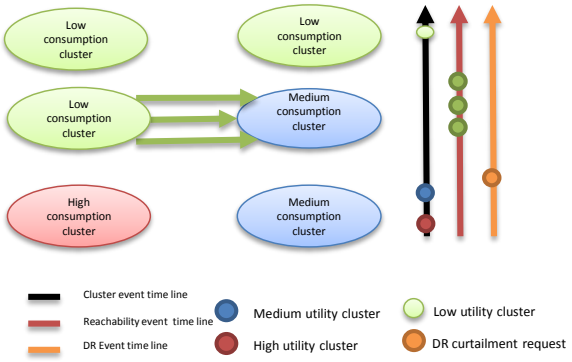


Figure 6. Power usage cluster detection and consumer social network reachability event detection to detect influencing groups.

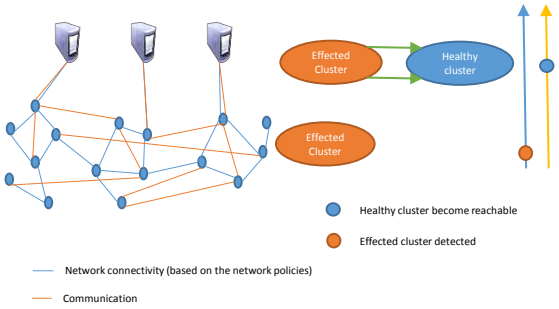


Figure 7. Warm infected cluster detection and prevention of infection propagation.

network of the cluster customers. We envision scenarios like after a demand response event a cluster of customers that have reduced their consumption influences through the social network another cluster of high consumption (cf. Fig. 6). This can be used to predict which group to be targeted for the demand response event.

Similarly, for the WAN we can consider the case of a denial of service attacks which can arise from the infected machines in a company's WAN. In this scenario we can look for the occurrence of infected clusters and reachability events from those clusters to other areas of the network. By correlating this with the occurrence of other infected clusters we can predict both the magnitude of infection and the likely hood it will happen in a particular sub-network (cf. Fig. 7). Finally this leads to preemptive measures against viral propagation.

B. Mapping the Use Case to the Architecture

1) Initial Data Loading: An initial data graph which contains the customer power grid connections and customer specific information is bulk loaded into the system . This initial graph instance will have customers as vertices and power distribution connections as edges. This process is similar for loading the WAN network structure information.

2) Stream Processing: In the smart grid scenario the system consists of three stream processor types: one for processing the power consumption events from the customers – which will emit graph events of attribute changes; another for processing the social network communication/network communication and emit graph events based on the communication between consumers/network nodes and also on their profile updates. This includes edge change updates and vertex attribute updates. Lastly, there is a type of stream processor which does not accept any input stream, but instead its task is to explore old communication links and emit edge removal events.

The WAN scenario contains only to stream processor types, for processing events related to package sent/receive, and for exploring old communication links and remove unused edges.

3) Continuous Graph Queries: For both cases we have two CGQs where one query depends on the other: (1) a cluster detection query which will detect cluster creations, merging of clusters and cluster disappearances; and (2) a reachability detection query which looks for reachability changes between various utility clusters. Given the causality motif only reachability events occurring after a cluster event will be looked for. Next we focus on the smart grid use case but a similar query can be devised for the WAN scenario. We need to monitor reachability events between low power consumption clusters to high power consumption clusters, followed by the decrease in consumption of the high power consumption cluster:

DETECT Reach(source:{name=C1.}, sink:{name=C2.*}) FOR 24h WHERE edge:{message} AND DETECT Cluster(sum(node.usage)* $< C$ *, diameter* $< d$ *) FOR (24h) WHERE edge:*$\{type = power\}$ AS C1 AND DETECT Cluster(sum(node.usage) $\geq C$, diameter $<$ d) FOR (24h) WHERE edge:$\{type = power\}$ AS C2;
DETECT Evolved(C2) WHERE sum(node.usage) $< C$ *AND SUCCEEDS C2*;

In the two queries *Evolved*, *Reach* and *Cluster* are predefined graph patterns.

4) Complex Graph Events and Analytics: In the smart grid based usecase the system will emit two main CGEs, namely cluster related events which emit events regarding cluster creations, merging of clusters and cluster disappearances, and reachability events which detect rechability changes between these clusters. An application that predicts power utility demand and send curtailment requests can subscribe to the these event streams by issuing CGQs to our system. These event streams can be used to predict new power demands.

In the WAN use case the system also triggers two CGEs, namely cluster related events which emit events regarding possibly infected clusters – based on their amount of communication to the servers –, and another to detect probable reachabilities from a suspicious cluster to a healthy cluster. By studying these events and their causal relation system administrators can take preventive actions.

VII. Future Research Directions

To enable the proposed model and architecture we have identified a set of research challenges that need to be addressed. They open a new range of future research directions that will be undertaken following this paper.

Scalable Fault Tolerant Data Stream Processing. To address the real time constraints of the streaming graph analytic Workers need to adapt to the input rate so that the CGEs are detected with a minimum latency from their occurrence. For this we require an elastic platform capable of both horizontal and vertical scaling. Both are enabled by the elastic buffer and require an efficient partitioning of the entire graph so that inter-partitioning communication is minimized and the computational effort is balanced across nodes. Smart state replication techniques, load balancing policies, availability mechanisms and machine learning techniques are required to enable a buffer based elastic forecasting approach for scaling streaming graphs.

Incremental Distributed Graph Stream Processing Algorithms and Abstractions. Real-time analytics on streaming graphs requires incremental algorithms which process graph updates and analyze them based on the CGQs. Some initiatives have been taken in this area [16] [19] but none of them are distributed or cater for our sophisticated graph event model. These algorithms need to be executed in a distributed manner due to time constraints and the dynamic topology and size of the graph. None of the existing graph processing programming abstractions provide capabilities for distributed and fault tolerant graph stream processing. This raises the need for a new distributed and fault tolerant graph stream processing abstractions.

Asynchronous Execution Models. Incremental streaming graph analytics require asynchronous message passing between the distributed Workers in order to reduce execution time and load balance. The asynchronous model poses additional issues than those encountered by traditional bulk synchronous models [5] [6]. These include synchronization between Workers, correlating the analytics rate to the buffer update rate, detecting general execution patterns for which this model can be optimized, etc.

Dynamic Re-Partitioning. As explained in Sect. IV the system continuously creates graph instances. Each instance is partitioned so it can be processed by a single machine. Due to the dynamic nature of the graph and because of fluctuations in input rate, volume and Worker load periodic re-assignments are needed. This raises the need for intelligent dynamic graph partitioning algorithms. These algorithms need to consider current load distribution, inter-Worker communication and behavior of the event buffer. Even though dynamic repartitioning has been discussed in literature [22], algorithms which consider this level of detailed objectives and input parameters were not addressed.

Distributed Data Structures and Indexes for Time Evolving Attributed Relational Graphs. When considering continuous data streams we need to design scalable data structures to handle the volume and velocity of the data. Research such as that conducted in [14] has mainly focused on large scale, high performance multicore machines. In order to enable fast processing of streaming graphs fast distributed data structures for storing the graphs and fast distributed indexes to enable CGQs are needed.

Efficient Translation Layer between high level abstract graph queries in CGQL and the low level graph constructs. CGQL allows users to capture a wide range of predefined or custom CGEs. Depending on their complexity and semantics they require a semantic match-making layer to be translated in specific graph elements such as vertices, edges and attributes. Domain specific ontologies need to be identified in this direction. The layer needs to perform the translation with minimum latency and can be incorporated in the Worker to reduce the communication overhead.

VIII. Conclusions

Due to the high velocity graph structured data streams generated by emerging domain specific applications the big data analytics domain experiences a shift in focus towards streaming graph analytics. This demands faster analytic models, architectures and algorithms which can provide analytic results in real-time. In this paper, we position ourselves in this context and look at the possible solutions and challenges in designing models, architecture and languages to enable real-time streaming graph analytics. We showed the applicability of the proposed model by presenting some example applications and mapping them to the proposed architecture and query language. Finally we described several research problems needed to be addressed in order to enable this kind of model and architecture.

ACKNOWLEDGMENT

This work was supported by the DARPA XDATA program. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. Authors would like to thank Charalampos Chelmis, Alok Kumbhare and Anand Panangadan for their input and feedback in preparing this article.

REFERENCES

[1] M. Cox and D. Ellsworth, "Application-controlled demand paging for out-of-core visualization," in *Proceedings of the 8th Conference on Visualization '97*, ser. VIS '97. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997, pp. 235–ff. [Online]. Available: http://dl.acm.org/citation.cfm?id=266989.267068

[2] "Big data," http://www.gartner.com/it-glossary/big-data/, accessed: 2014-01-13.

[3] "Storm distributed and fault-tolerant realtime computation," http://storm-project.net/, accessed: 2014-01-07.

[4] "Amazon kinesis," http://aws.amazon.com/kinesis/, accessed: 2014-01-07.

[5] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[6] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, , and V. Prasanna, "Goffish : A sub-graph centric framework for large-scale graph analytics," USC, Tech. Rep. 13-936, 2014, http://arxiv.org/pdf/1311.5949v1.pdf.

[7] C. C. Aggarwal and H. Wang, *Managing and mining graph data*. Springer, 2010, vol. 40, ch. Query language and access methods for graph databases.

[8] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[9] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 505–516. [Online]. Available: http://doi.acm.org/10.1145/2463676.2467799

[10] U. Kang, C. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on*, 2009, pp. 229–238.

[11] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *CoRR*, vol. abs/1204.6078, 2012.

[12] S. Asur, S. Parthasarathy, and D. Ucar, "An event-based framework for characterizing the evolutionary behavior of interaction graphs," *ACM Trans. Knowl. Discov. Data*, vol. 3, no. 4, pp. 16:1–16:36, Dec. 2009. [Online]. Available: http://doi.acm.org/10.1145/1631162.1631164

[13] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 85–98. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168846

[14] D. Ediger, R. McColl, J. Riedy, and D. Bader, "Stinger: High performance data structure for streaming graphs," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, 2012, pp. 1–5.

[15] D. Ediger, J. Riedy, D. A. Bader, and H. Meyerhenke, "Tracking structure of streaming social networks." IEEE, pp. 1691–1699. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6009035

[16] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu, "Incremental graph pattern matching," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 925–936. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989420

[17] A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. Miller, "Towards efficient query processing on massive time-evolving graphs," in *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2012 8th International Conference on*, 2012, pp. 567–574.

[18] R. McColl, O. Green, and D. Bader, "A new parallel algorithm for connected components in dynamic graphs," in *IEEE International Conference on High Performance Computing*. [Online]. Available: http://www.stingergraph.com/data/uploads/papers/dynamiccomponents.pdf

[19] P. R. Mullangi and L. Ramaswamy, "Scissor: Scalable and efficient reachability query processing in time-evolving hierarchies," in *Proceedings of the 22Nd ACM International Conference on Conference on Information & Knowledge Management*, ser. CIKM '13. New York, NY, USA: ACM, 2013, pp. 799–804. [Online]. Available: http://doi.acm.org/10.1145/2505515.2505732

[20] J. Riedy, H. Meyerhenke, D. Bader, D. Ediger, and T. Mattson, "Analysis of streaming social networks and graphs on multicore architectures," in *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, 2012, pp. 5337–5340.

[21] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[22] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," Stanford InfoLab, Tech. Rep., 2013.

[23] T. Bass, "Mythbusters: Event stream processing versus complex event processing," in *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, ser. DEBS '07. New York, NY, USA: ACM, 2007, pp. 1–1. [Online]. Available: http://doi.acm.org/10.1145/1266894.1266896

[24] D. Robins, "Complex event processing," 2010.

[25] M. P. Consens and A. O. Mendelzon, "Graphlog: A visual formalism for real life recursion," in *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '90. New York, NY, USA: ACM, 1990, pp. 404–416. [Online]. Available: http://doi.acm.org/10.1145/298514.298591

[26] R. H. Güting, "Graphdb: Modeling and querying graphs in databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 297–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=645920.672980

[27] D. H. Kim, I. D. Yun, and S.-U. Lee, "A new attributed relational graph matching algorithm using the nested structure of earth mover's distance," in *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, vol. 1, 2004, pp. 48–51 Vol.1.

[28] "Twitter," https://twitter.com/, accessed: 2014-01-07.

[29] "Friendfeed," http://friendfeed.com/, accessed: 2014-01-07.

Cost-efficient and Resilient Job Life-cycle Management on Hybrid Clouds

Hsuan-Yi Chu
Computer Science Department
University of Southern California
Los Angeles, CA 90089, U.S.A.
Email: hsuanyi@usc.edu

Yogesh Simmhan
Supercomputer Education and Research Center
Indian Institute of Science
Bangalore 560012, India
Email: simmhan@serc.iisc.in

***Abstract*—Cloud infrastructure offers democratized access to on-demand computing resources for scaling applications beyond captive local servers. While on-demand, fixed-price Virtual Machines (VMs) are popular, the availability of cheaper, but less reliable, *spot VMs* from cloud providers presents an opportunity to reduce the cost of hosting cloud applications. Our work addresses the issue of effective and economic use of hybrid cloud resources for planning job executions with deadline constraints. We propose strategies to manage a job's life-cycle on spot and on-demand VMs to minimize the total dollar cost while assuring completion. With the foundation of *stochastic optimization*, our reusable table-based algorithm (RTBA) decides when to instantiate VMs, at what bid prices, when to use local machines, and when to checkpoint and migrate the job between these resources, with the goal of *completing the job on time and with the minimum cost*. In addition, three simpler heuristics are proposed as comparison. Our evaluation using historical spot prices for the Amazon EC2 market shows that RTBA on an average reduces the cost by 72%, compared to running only on on-demand VMs. It is also robust to fluctuations in spot prices. The heuristic, H3, often approaches RTBA in performance and may prove adequate for *ad hoc* jobs due to its simplicity.**

I. Introduction

Cloud computing has become a first-class resource platform for application service providers to host their services and complement their local computing infrastructure. The lower barrier to entry, both in the initial infrastructure cost and the service-based middleware tooling, has made clouds popular for enterprise and scientific applications [4]. Public cloud service providers such as Amazon Web Services (AWS) and Microsoft's Azure offer access to compute resources through Virtual Machine (VM) instances with diverse capabilities, and to persistent storage services.

Cloud providers follow a *pay-as-you-go* pricing model, where customers pay for the resources they acquire on-demand, in well-defined time and resource increments (e.g. hourly increments, small/medium/large VMs), using a *static price* list. Hence acquiring and releasing resources elastically is important to maximize the utility. While this elasticity is easy to leverage for repetitive, stateless jobs such as web services, larger applications can use this elasticity to pick a VM of the right size and use it exclusively rather than compete with other jobs on the same VM. This puts the focus on resource allocation for a single job rather than scheduling multiple jobs on few resources.

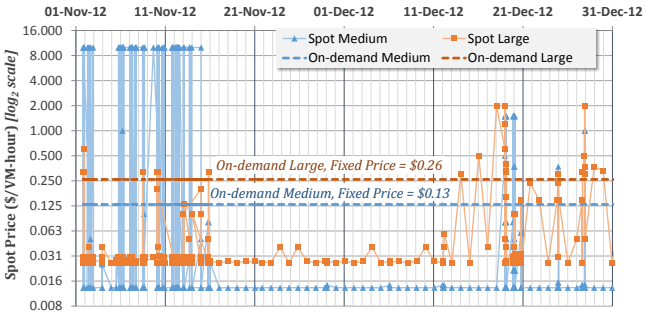
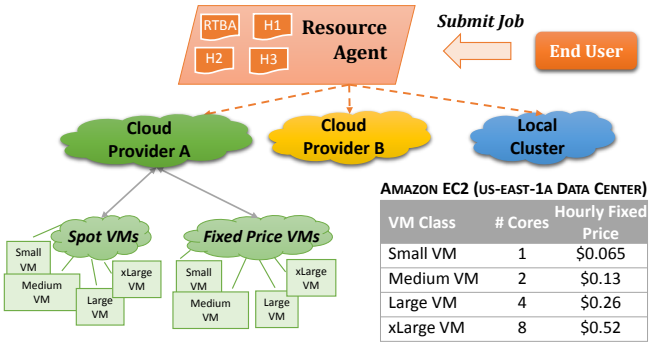


Figure 1: Amazon EC2 spot prices of medium and large instances at *us-east-1a* data center in Nov and Dec, 2012. *Fixed-price* on-demand VMs are also shown for comparison. Note that the spot prices of the medium VMs have significant fluctuations in Nov while being more stable in Dec.

Recently, cloud providers such as Amazon are offering *spot priced* VMs. These are typically spare data center capacity offered at low prices to enhance the data center's resource utilization. Customers place their *bids* for VMs on the spot market at a price level of their choosing [1]. The cloud provider periodically publishes the actual spot prices at which they will offer these VMs. Customers whose bids are greater than the published spot prices will be given the VMs. The cloud provider can vary the spot prices over time, sometimes within minutes. Customers who have acquired a spot VM pay the current spot price as long as it is less than their bid price. However, when the spot price increases beyond their bid, the VM is immediately revoked from the customer. This is termed as an "out-of-bid event", and *all the applications, state and data present in that VM are lost*. Thus, while spot VMs are usually cheaper than fixed-price VMs and perform equally well, they are susceptible to revocation due to the pricing mechanism of the cloud provider which in itself is not a true market-based model. For instance, in Fig. 1, the prices for medium-sized spot VMs are usually $0.014/hour as compared to $0.13/hour for the fixed-price ones, but spike to over $8/hour, thus revoking literally all medium spot VMs. As a result, long running applications that pay a cumulative higher cost for fixed-price VMs and



AMAZON EC2 (US-EAST-1A DATA CENTER)

VM Class	# Cores	Hourly Fixed Price
Small VM	1	$0.065
Medium VM	2	$0.13
Large VM	4	$0.26
xLarge VM	8	$0.52

Figure 2: The resource agent accepts jobs from users with specified deadline constraints. The agent plans strategies to provision resources and manage checkpoint/migration on a hybrid cloud to finish the job by the deadline with the minimum rental cost.

2014/2/1

would benefit from using cheaper spot VMs will need to be resilient, using simple *ab initio* restart after losing their prior work, or through checkpoint and resume mechanisms.

Problem. We refer to the diverse and distributed computing resources available to a customer: fixed-price on-demand VMs [1], spot VMs, and limited capacity on "free" local servers or private clouds they own, as a *hybrid cloud* environment (Fig. 2). Cloud VMs further have different VM sizes and pricing based on computing capabilities. A job with an expected runtime (core-hours) and a deadline for completion has to make the best use of the hybrid cloud to minimize total cost while ensuring completion within the deadline. Our proposed resource agent *makes decisions on the life-cycle of this job* – which resource to start on, at what price to bid for spot VMs, when to checkpoint, when to migrate and resume, and when to restart – to meet the objectives. In particular, the challenge arises from intelligently using spot VMs at the right bid price to balance total cost against the deadline, and mitigating the impact of out-of-bid events through job life-cycle management. This is a novel problem that confronts real applications, with the ability to maximize the economic utility of hybrid resources.

Results. With the foundation of *stochastic optimization*, we propose a **Reusable Table-Based Algorithm (RTBA)** which facilitates the resource agent's decision-making for a given job's life-cycle on hybrid resources. The algorithm is reduced to *Bellman's equation* to meet the deadline and cost minimization objective, proving the *minimum* expected resource cost. We also propose three simpler heuristics (**H1–H3**) for making these decisions that may prove adequate and cost effective for *ad hoc* jobs. Our simulation study using different job characteristics and 5 months of Amazon EC2 Spot VM pricing demonstrates the following:

- Our RTBA approach reduces the resource cost by an average of 72%, compared to running purely on fixed-price on-demand VMs, for the job workload we study.
- All three heuristics H1–H3 offer less cost savings than RTBA with average improvements of 43%, 53% and 67%, respectively, compared to purely fixed-price VMs. The heuristic H3 often performs close to RTBA and may be more practical due to its simplicity.
- RTBA uses historical spot prices to develop a price model which offers spot price estimates [2]. We empirically show the robustness of RTBA to different weakly reliable price models. The RTBA's cost performance varies narrowly for different models but is always better than on-demand VMs and the heuristics.

The rest of the paper is structured as follows: we present related work in § II, define the hybrid cloud system and job model in § III, formalize the problem mathematically in § IV, develop optimal and heuristic-based solutions in § V and § VI, summarize the results of our simulation study in § VII, and offer our future directions and conclusions in § VIII.

II. Related Work

Public cloud resources cost real money. Hence, from a cloud user's perspective, it is important to minimize *monetary cost*, which is distinct from the minimization of *resource usage* [24]. The non-uniform pricing of spot VMs offers the possibility of using more (but transiently cheaper) resources for a cost effective solution. As a result, existing resource minimization algorithms that may suffice for fixed-price VMs are inadequate for variable-priced spot VMs. Our work lies in the intersection of several research areas.

Hybrid Clouds. Cloud bursting is a term used when an organization uses in-house servers to support normal application needs and requests additional cloud resources during workloads spikes [6], [9], [17]. We consider an analogous but inverse model where jobs are primarily queued and run on clouds where they have the option of choosing from different VM types, but can also access limited off-cloud local servers. This predicates the use of clouds for *normal operations at scale* and sparse (but free) local resources as a backup. One distinction here is that, instead of an overhead to deploy jobs from local servers to the cloud, there is a cost for moving the job and its data out of the cloud.

Hosting scientific applications on Amazon EC2 has attracted significant attention [10], [13]. Some of them consider cost and deadline factors in provisioning resources [16], [15], [11]. For instance, [11] considers a similar problem of minimizing user-payments with guaranteed task deadlines. However, they do not consider spot VMs which can potentially be even cheaper but also requires novel approaches to ensure job resilience upon VM revocation.

Spot Pricing. The VM spot market was first introduced by Amazon in Dec, 2009 to exploit their unused data center capacity [1]. Spot VMs are usually much cheaper than their

[1]Both 'fixed-price on-demand' and 'spot' VMs are, in effect, acquired on-demand. Our terminology is based Amazon AWS's convention. We use the terms 'fixed-price on-demand' and 'on-demand' VMs interchangeably.

fixed-price equivalents though they offer the same performance, but without guaranteed resource availability. Spot prices are influenced more by Amazon's usage of on-demand VMs and spare capacity rather than the spot bids received or spot VM workloads that are submitted. Some studies have analyzed historical spot prices in an attempt to reverse-engineer the pricing logic. For instance, [2] speculates that the prices are determined artificially by a random reserve price algorithm. Moreover, [8] and [22] employ *Markov chain models* to characterize the spot prices. Importantly, while [23] shows the difficulty of predicting the spot prices, they also demonstrate the effectiveness of probabilistic spot price modeling and *stochastic optimization*, which serve as the basis of our RTBA approach.

Resilience. While bidding at a higher price can reduce the chance of revocation, the peak spot prices are way higher than fixed-prices of VMs (Fig. 1), making spot VMs inherently unreliable. Hence, bidding and fault-tolerance mechanisms are dual problems, and optimal decision-making should jointly consider these two dimensions. Checkpointing is a common approach to enhance resilience of jobs on unreliable resources. There is rich literature on checkpointing for HPC applications with the overhead being measured and modeled [7], [14]. Costs of checkpointing and migration may be asymmetric. Checkpointing cost may be trivial for some applications [22], [23] while dominating for others [19]. So checkpointing must be done judiciously to mitigate the overhead and additional resource costs.

[23] proposes an optimization problem which uses the *stochastic resource rental model* to plan resource provisioning on spot and fixed-price VMs. However, a key simplifying assumption in their model is that when an out-of-bid event occurs, the state of the job can be immediately migrated to an available on-demand VM. This is unrealistic as out-of-bid events are not notified in advance and occur instantaneously. Our proposed strategy addresses this lacuna. Specifically, once a VM is revoked, the job resumes from the last available checkpoint. In addition, the costs of checkpointing and migration are modeled [2], and the *startup time* for acquiring VMs is considered – these matter in practice. In other literature, out-of-bid events are handled either through checkpointing, migration and replication strategies [21], [20], or through bidding schemes [3], [18], [22]. Nevertheless, our work is unique since we co-design the bidding price and fault-tolerance decisions, which together minimize the cost while meeting the job's deadline requirement.

III. System and Job Models

A user submits a job J to our resource agent which makes life-cycle management decisions for it. Both the job and the agent are assumed to be in the cloud initially. The characteristics of the job J are described as a tuple $J(C, D, T_{dl})$, where D is the input data size, C is the number of core-hours of computation required, and T_{dl} is the deadline by which the job has to be completed. The followed notations are summarized in Table I. Further, we assume that the job speeds up linearly with the number of CPU cores present on a machine (i.e., it takes a 1-core small VM twice as long as a 2-core medium VM to finish a job).

Table I: Conventional Notations

Variable	Description
C	Job's Compute requirement (core-hours)
D	Job's input data size (bytes)
T_{dl}	Job's deadline constraint (hours)
$J(\cdot)^a$	Characteristics of the job J
t	Logical time index
a_t	Action taken at t
$\pi_{t'}$	Policy, a series of actions from $t=0$ to $t=t'$
P_t^i	Spot price of the VM class i at t
$\mathbf{P}_t$	Set of spot prices for all VM classes at t
$c(t)$	Job's progress at t [b]
$c'(t)$	Job's residual compute after the last checkpoint
$x(t)$	Job's state at t
$A(x(t))$	Set of available actions for state $x(t)$
$v(t)$	Class of the VM that this job runs on at t
$\rho(v(t))$	CPU cores available in the VM of class $v(t)$
Δt	Index of the time-slot within a whole VM hour
$S_t(\cdot)$	Status of the job at t

[a] The notation $(\cdot)$ denotes the omitted argument of a tuple.
[b] This is measured as the residual computing that is still needed at t.

Based on offerings from contemporary cloud providers, the resource types we consider are: on-demand VMs, spot VMs and off-cloud local servers. VMs further offer multiple classes, differentiated by the number of cores (see Table in Fig. 2). Each class of on-demand VMs has a fixed hourly price, depending on the number of cores; fractional hours used are rounded up to whole hours. Spot VMs are priced on a different policy: (1) The price of each class varies independently, with time and orthogonal to the user's requests. Notably, the larger VMs with more cores are not necessarily costlier than the smaller one (see Fig. 1). (2) Spot VMs have to be bid for at a specified bid price. As long as the user's bid is higher than the current spot price, the VM is available to the user. (3) In case of out-of-bid events, where the spot price increases beyond the bid, the VM is immediately revoked and terminated with loss of all state; the partial hour used is not be billed. Otherwise, billing is by the hourly usage, as for fixed-price VMs. Both on-demand and spot VMs are associated with a start-up time – real clouds do not provision VMs instantaneously. Utilizing off-cloud local machines does not incur additional expense. However, transferring data from the cloud to the local machines costs network bandwidth that is billed proportional to the job's data size, D.

[2] While both checkpointing and migration result in overhead times, migration can incur additional network charge if the job and its data are moved out of the cloud; often, in-cloud bandwidth is free while out-of-cloud bandwidth is billed.

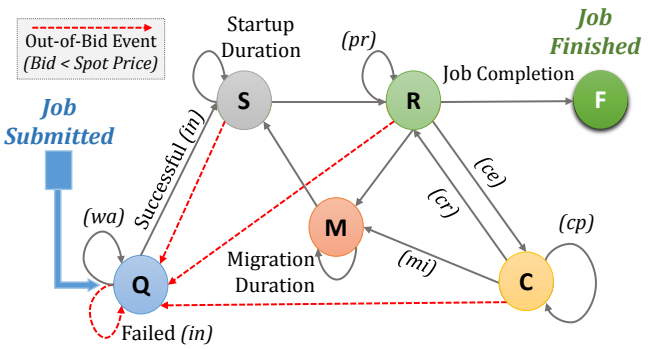


Figure 3: The state transition diagram for a job's life-cycle managed by the agent. Edges are labeled with actions taken by the agent from the set A, or forced by the cloud provider.

Both checkpointing and migration use a resource's time during which the job cannot progress. The times taken to perform checkpointing and migration are given as $T_C = \epsilon(c)$ and $T_M = \frac{D}{BW}$, respectively, where $\epsilon(c)$ is a function of the job-progress [3], D is the data size, and BW represents the *bandwidth* associated with the direction of data transfer (i.e., intra-cloud, local machine→cloud, or cloud→local machine), each of which can be different. While some prior works have ignored the cost of checkpointing [22], [23], in practice it can dominate application runtimes [19], particularly when applications are checkpointed more frequently if the *mean-time between failure* of the host machine is small. We incorporate a swathe of parameters in our framework to capture real system behavior and offer a wide solution space. These parameters can be pruned to balance accuracy, ability to collect the relevant information, and model complexity.

The time taken for checkpointing, migration and VM startup [4] can significantly undermine the economic benefits indirectly. Specifically, during these operations, a job cannot progress in its execution, and they cause the deadline to virtually "shrink". As can be imagined, an imminent deadline might preclude the use of cheaper spot VMs and force us to use more expensive resources, such as reliable on-demand VMs or larger sized VMs, to ensure on-time job completion. Thus, an intelligent agent, managing a job's life-cycle, must plan a series of actions that lead to the job's completion by deadline, and with the minimum total cost.

To begin with, we define the following terminology. The state of a job at time t is encoded by $x(t)$. A job can reside in one of the six states, detailed later. The *action*, a_t, applied at time t refers to the decision the agent makes at that instant, in response to the observed conditions. The available actions are given by the set $A = \{in, pr, ce, cp, cr, mi, wa\}$, which corresponds to initiating/bidding for an instance (*in*), processing the job (*pr*), starting to checkpoint (*ce*), continuing checkpointing (*cp*)[5], resuming from a checkpoint (*cr*), migrating the job (*mi*), and waiting for spot price to drop (*wa*). The *policy*, $\pi_{(T_{dl}-1)}$, is a sequence of actions $\{a_0, a_1, ..., a_{(T_{dl}-1)}\}$ from $t = 0, \ldots, (T_{dl} - 1)$ [5], which starts from the job's initial state, through a sequence of intermediate states, to the completion state.

We introduce the *transition diagram*, shown in Fig. 3, to describe the life-cycle of a job. It is assumed that the time-axis is divided into a series of time-slots, and actions are chosen at the start of a time-slot. There are five active states, *(1)* ***Q****uiesced, (2)* ***S****tartup, (3)* ***R****unning, (4)* ***C****heckpointing, (5)* ***M****igrating and (6)* ***F****inish*, that are indexed by a number, $x(t) \in [1, 6]$. As is shown in the figure, the available actions at any given state is a subset of A, thus denoted by $A(x(t))$.

As is shown in Fig. 3, a job starts in the quiesced state. This state is also entered if under-bidding, out-of-bid events or strategic waiting (for the spot price to drop) occurs. The job is in the startup state while it waits for a VM to be provisioned. This is used to model VM startup cost. A job that is being processed is in the running state. When a job enters the checkpointing state from the running state, it stays there while the checkpoint is completed. Note that both the running and checkpointing states can be interrupted by an out-of-bid event. In the migration state, the job transfers data to a new machine and prepares to resume from that point. This state captures both the migration time and data-transfer cost. Lastly, the finish state is reached upon the job's completion. A job is assumed to remain in a state for at least one time-slot. Further, the running and checkpointing states incur VM rental costs, including partial hours, that are tracked by a parameter Δt (illustrated in the next section).

IV. Mathematical Model

The spot price of each class of spot VM fluctuates along the time-axis, forming a time-series which is described as a *spot price stochastic process* (*SPSP*). Moreover, since different classes of spot VMs are priced independently [20], each class is modeled using an independent SPSP. Markov chain has been exploited in existing literature to model the spot prices [8], [22]. Similarly, we model each SPSP as a Markov chain P_t^i, where i refers to a class of spot VMs, and t represents the time:

$$\begin{cases} Pr\left\{P_t^i = a \middle| P_{t-1}^i = a\right\} = (1 - \alpha_i) \\ Pr\left\{P_t^i = b \middle| P_{t-1}^i = a, P_t^i \neq a\right\} = f_P^i\left(b | P \neq a\right) \end{cases} \quad (1)$$

where a and b refer to two spot prices, α_i captures the rate of price fluctuations, and $f_P^i(\cdot|P \neq a)$ is a conditional probability distribution that describes the price transition pattern [8].

Given a job tuple $J(C, D, T_{dl})$, the resource agent needs to determine the policy to complete this job by the deadline. The *status* of a job at t is characterized as a tuple

[3] $\epsilon(c)$ models the checkpointing time as a function of the current job progress. For example, the model in [14], [7] can be used as $\epsilon(c)$.

[4] VMs take time to deploy and boot up. This overhead can be punitive if one switches frequently between VMs.

[5] Before checkpoining is completed, it is possible to quit checkpointing and resume at the state where checkpointing has not been started yet.

$S_t(c(t), c'(t), x(t), v(t), \Delta t))$, where $c(t)$ is the amount of remaining compute requirements (core-hours), $c'(t)$ is the amount of remaining compute requirement *since the last checkpoint* (core-hours), $x(t)$ is the job's state, $v(t)$ is the class of the VM this job is residing on, and Δt is the index of the time-slot within a complete hour. As stated in § III, this parameter tracks whether a complete VM hour is finished, and users are charged once the whole hour is reached. Note that despite the similar terminology, "status" $S_t(c(t), c'(t), x(t), v(t), \Delta t))$ and "state" $x(t)$ are two different concepts.

The life-cycle of a job corresponds to a route, connecting edges between nodes in Fig. 3. Here, we describe mathematically as to how the job's status, $S_t(c(t), c'(t), x(t), v(t), \Delta t))$, is updated with the transition between nodes (i.e., states) through edges (i.e., chosen actions) in the state transition diagram. When an out-of-bid event or a migration occurs, we update $c(t+1) = c'(t)$, which means the job's progress is lost, and it has to resume from the last checkpoint. On the other hand, when a checkpoint is completed successfully, we update $c'(t+1) = c(t)$, which implies that if an out-of-bid event occurs later, the job can resume from this new checkpoint. When the action *pr* is chosen, the job progress $c(t)$ is updated as

$$\begin{aligned} c(t+1) &= \left(1 - I_{\{out-of-bid\}}\right) \times \left[c\left(t\right) - \rho\left(v\left(t\right)\right)\right] \\ &+ I_{\{out-of-bid\}} \times c'\left(t\right) \end{aligned}$$

where $I_{\{.\}}$ is an *indicator function*, which is equal to 1 if the condition holds, and $\rho(v(t))$ gives the number of cores in the VM $v(t)$ at time t.

Given the mathematical definition of the status update, the objective function can be formulated. Due to the deadline constraint, we impose the constraint $c(T_{dl}) = 0$. A policy $\pi_{(T_{dl}-1)}$, a sequence of actions $\{a_t\}$, where $t = 0, ..., (T_{dl} - 1)$, is chosen such that:

$$W_{(T_{dl}-1)} = \min_{\pi_{(T_{dl}-1)}} \mathbb{E}\left[\sum_{t=0}^{T_{dl}-1} u\left(a_t, S_{(t+1)}\right) \middle| S_0 ... S_t, \mathbf{P}_0 ... \mathbf{P}_t\right]$$

$$\text{subject to} \quad c(T_{dl}) = 0 \tag{2}$$

where $W_{(T_{dl}-1)}$ is the expected cost spent from $t = 0$ to $t = T_{dl}$, resulting from the policy $\pi_{(T_{dl}-1)}$. The operator $\mathbb{E}$ refers to the expectation over the spot prices, S_t and $\mathbf{P}_t$ are the job's status and spot prices at t, respectively, a_t is the action taken at t, and $u\left(a_t, S_{(t+1)}\right)$ denotes the cost function of that action and the resulting status $S_{(t+1)}$. Note that the resulting status depends not only on the taken action a_t but also on the (probabilistic) spot prices $\mathbf{P}_{(t+1)}$, over which the expectation is performed.

V. Optimal Solution using RTBA

In this section, the optimal solution for Eq. 2 is found with the formalism of Bellman equation, and the solution then inspires our Reusable Table-Based Algorithm (RTBA) to effectively plan the policy.

A. Optimal Decision-making

Bellman's equation represents the value of an objective function at a decision point in terms of the sum of an action's cost and the value of the residual objective function that results from that action [5]. Given the Markov chain assumption Eq. (1) along with the *law of iterated expectations*, Eq. (2) is revised as:

$$\begin{aligned} W_{(T_{dl}-1)} &= \min_{\pi_{(T_{dl}-1)}} \mathbb{E}\left[\sum_{t=0}^{T_{dl}-1} u\left(a_t, S_{(t+1)}\right) \middle| S_t, \mathbf{P}_t\right] \\ &= \min_{a_0} \mathbb{E}\left[u\left(a_0, S_1\right) | s_0, \mathbf{P}_0\right] + W_{(T_{dl}-2)} \end{aligned} \tag{3}$$

We arrive at Bellman's equation with finite horizon from $t = 0, \ldots, (T_{dl} - 1)$. This equation reduces the choice of a policy $\pi_{(T_{dl}-1)}$ to a sequence of choices of a_t. Note that the subscript of $W_{(T_{dl}-1)}$ should not be interpreted as time. Instead, it refers to the length of the policy (i.e., the number of actions), which leads to the cost $W_{(T_{dl}-1)}$. In particular, given the current job state at time t as $x(t)$, we choose, from the set of applicable actions $A\left(x\left(t\right)\right)$ the action and its corresponding sub-policy whose sum is the minimum. Formally,

$$a_t = \arg\min_{a_t} \mathbb{E}\left[u\left(a_t, S_{(t+1)}\right) \middle| s_t, \mathbf{P}_t\right] + W_{(T_{dl}-t-2)} \tag{4}$$

which suggests a recursive relationship for the optimal decision-making routine.

B. Action Profile

We illustrate in Fig. 4 how the resource agent chooses the actions in response to spot price fluctuations, termed as *action profiles*. The test period of the spot prices is chosen from 00:00 a.m. to 10:00 a.m. on Dec, 19, 2012, when the medium spot VM (dotted blue line) has more fluctuations in the first half of the observation period while the large spot VM (dashed green line) has more fluctuations in the second half. Further, the SPSP model is based on the historic spot prices from Aug, 2012 to Oct, 2012, the job's compute requirement (C) is set to 8 core-hours, and the deadline (T_{dl}) is chosen as 10 hours.

In Fig. 4, the chosen actions for the job are represented as a solid orange line, and the out-of-bid events and checkpointing are identified with ✘ and ▲ signs, respectively. In the realistic scenario, VMs are initiated after a startup delay; this is visible in the brief time spent in the *Startup* action prior to the job *Running* on a VM. There are three out-of-bid events during the job's life-cycle – the first two occur while the VM is starting and the third one revokes the large spot VM from the resource agent, causing the progress made after the second checkpoint to be lost.

Since the large spot VMs are cheap at the beginning, the resource agent selects them initially to run the job.

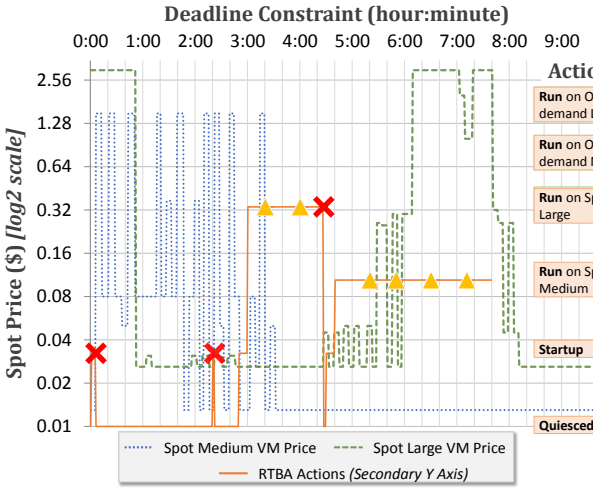


Figure 4: The X-axis shows the job's life time. The primary Y-axis shows the current spot prices of medium and large VMs (dotted blue and dashed green lines respectively). The secondary Y-axis shows various actions taken by the RTBA algorithm (solid orange line). The x and ▲ signs represent out-of-bid events and checkpointing.

The resource agent bids a large VM at 2:50 a.m. and then performs checkpoints of its progress. After an out-of-bid event strikes the large VM at 4:27 a.m., the resource agent has to re-bid for a new VM and resume the job from the previous checkpoint. Interestingly, this time the resource agent selects a medium spot VM since it predicts that the price of large VMs is likely to increase (as it does in the latter half of the test period). The more stable-priced medium spot VMs are used to finish the job at 7:40 a.m, with period checkpoints and well ahead of the deadline.

C. Reusable Table-Based Algorithm (RTBA)

As is shown in Eq. 4, the optimal action at any time can be determined if W_τ is evaluated in a *backward* fashion from the base case $\tau = 0$. Note that the subscript of W_τ refers to the length of the policy, which leads to the cost W_τ. W_0 represents the base case since it corresponds to the deadline, when no more actions need to be decided. Intuitively, the base case falls into one of two situations: completed jobs ($c(T_{dl}) = 0$) and incomplete jobs ($c(T_{dl}) > 0$). Since the latter case fails the deadline constraint, we set its $W_0 = \infty$. On the other hand, the former implies the job has finished at or prior to the deadline. Hence, $W_0 = 0$ if $\Delta t = 0$; $W_0 = p_t^{v(0)}$ if $\Delta t \neq 0$. This is because if a new VM-hour cycle has not yet started ($\Delta t = 0$), users are not charged. Else, users are charged according to the VM's price, $p_t^{v(0)}$.

Starting from the base case, one can obtain the optimal policy by solving Eq. 4 in a backward fashion, from $t = (T_{dl} - 1)$ to $t = 0$, for every combination of $(c(t), c'(t))$. We solve this using a *dynamic programming* framework, which stores the value of the objective function at $t = t'$ for the evaluation of the value at $t = (t' - 1)$. Thus, the complexity is $O(C^2 T_{dl})$.

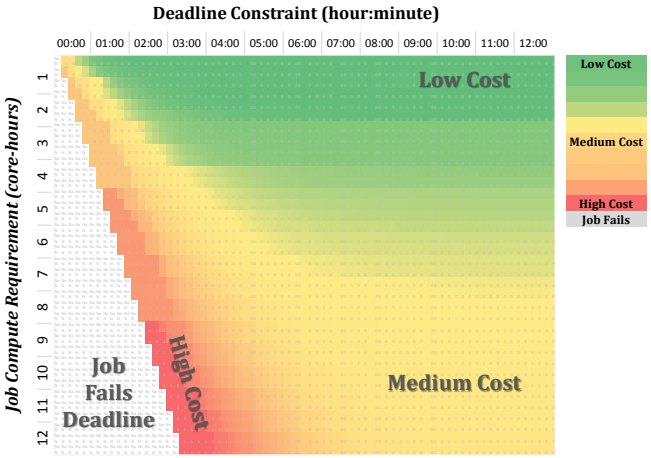


Figure 5: Heatmap showing expected cost of running jobs with different deadline constraints (X-axis, hours:mins) and compute requirements (Y-axis, core-hours). The costs are described by a color scale, green being cheap and red costly. The grey region in the bottom left indicates jobs with short deadlines that fail to meet the constraint.

Scanning and plotting the solution space as a *heatmap* in Fig. 5 offers visual insight. We build the SPSP model using historic spot price data from Aug, 2012 to Oct, 2012, and programmatically solve the optimization problem for diverse combinations of compute requirements and deadlines. The value of the objective function for different deadlines (X-axis) and compute requirements (Y-axis) is depicted using a color scale. This heatmap highlights the optimal trade-offs between compute requirements, deadlines and budgets, and helps answer the following questions: (1) *Given a budget and the job's compute requirement, what is the minimum deadline can one set?* (2) *Given the job's compute requirement and deadline, how much would one expect to spend?* and (3) *Given a job's compute requirement, where is the "sweet spot", where a small concession on the deadline offers a significant cost reduction (i.e., sharp color-transitions in the heatmap)?*

This solution inspires our Reusable Table-Based Algorithm (RTBA). Since a table is being constructed when we solve the optimization problem backwards, this table, termed as the *strategy table*, can be queried whenever an action needs to be decided. Notably, the strategy table is reusable for other jobs if their compute requirement C and deadline constraint T_{dl} fall within this table's bound. This is because if the job falls within this bound, there should be an entry in the strategy table which corresponds to this job, and which was solved as a sub-problem when constructing the table using dynamic programming. Essentially, RTBA constructs a strategy table offline, and when an action is to be decided, performs a simple table look-up. The table can be reused and expanded if larger bounds are necessary. Importantly,

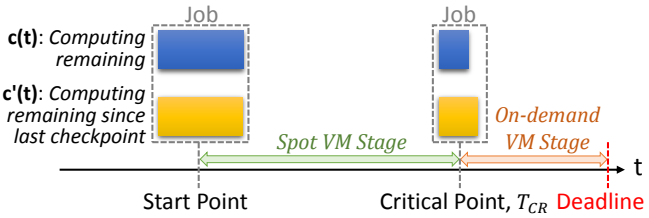


Figure 6: The value of residual compute requirement, $c(t)$, is represented on top as a blue block, and the residual compute required since the last checkpoint, $c'(t)$, is represented below as a yellow block. At the start, $c'(t) = c(t)$ since no checkpoints have been made. For H1, $c'(T_{CR}) = c(0)$, and for H2, $c'(T_{CR}) = c(T_{CR})$, where T_{CR} remains constant. For H3, $c'(T_{CR}) = c(T_{CR})$, but T_{CR} is pushed forward.

while the construction of the strategy table depends on the modeled SPSP, whose pattern might change over time [2], we experimentally observe that RTBA is robust over time due to the probabilistic approach to spot price modeling [23]. Therefore, near-optimal solutions can be achieved even without frequently reconstructing the strategy table to include recent spot prices.

VI. Heuristic-based Solutions

While RTBA offers an optimal solution, constructing the strategy table can be costly, particularly if the number of jobs is too small to amortize the construction cost, or the job diversity is too large to enable reuse. We propose three heuristics to facilitate agile decision-making for *ad hoc* jobs. These heuristics also offer baselines of increasing sophistication to compare against RTBA.

Spot prices are generally much cheaper than their on-demand counterparts. Intuitively, the more one uses spot VMs, the less they are going to spend overall. Essentially, our heuristics process a job using spot VMs until a potential out-of-bid event would threaten the job's feasibility to complete on-time. We define *critical point*, T_{CR}, as the time beyond which a job cannot be completed after resuming from its checkpoint. This illustrated in Fig. 6.

$$T_{CR} = \frac{c'(t)}{\rho\left(v\left(t\right)\right)} \tag{5}$$

Thus, the critical point is a function of $c'(t)$, the job's remaining compute requirement at the last checkpoint, and $\rho\left(v\left(t\right)\right)$,the processing capability of its current (spot) VM. The critical point locates the transition from spot to on-demand VMs. We refer to the time prior to the critical point as *spot stage* and to that after the critical point as *on-demand stage*. The three heuristics vary in when they perform checkpoints. We define the *naïve critical point* by setting $c'(t) = C$ in Eq. 5. The naïve critical point is conservative in choosing the transition point, and ensures that the job can *restart* and complete even if no checkpoints have been performed in the spot stage.

H1:Two-Stage Algorithm. This heuristic uses only the spot VMs prior to the naïve critical point, does not perform any checkpoints. It *restarts* the job on the on-demand VMs if the job is not completed by the naïve critical point. This can lead to over-payments since the expenditure on the spot stage does not contribute to the job's completion if it did not finish before the naïve critical point.

H2:Boundary Checkpoint Algorithm. This heuristic is similar to H1 but attempts to transfer the progress made in the spot stage to the on-demand stage. When the naïve critical point is reached, this heuristic performs a checkpoint prior to switching to on-demand VMs, and resumes the job from that checkpoint. However, it is still vulnerable to out-of-bid events close to the naïve critical point.

H3:Rolling Checkpoint Algorithm. This heuristic tries to push forward the critical point to allow the job to continue running on spot VMs for longer. Due to the functional relationship between T_{CR} and $c'(t)$, we periodically checkpoint and update $c'(t)$. We define the *effective frequency*, F_{eff}, as the duration between two consecutive checkpoints. Ideally, F_{eff} should allow the job to progress sufficiently to outweigh the cost of checkpointing. For simplicity, we fix F_{eff} to be equal to the average lifetime of a particular spot CM class multiplied by its number of cores. A more sophisticated approach can set F_{eff} as a function of the current spot price.

VII. Results

A. Simulation Setup

We consider five machine types, ⟨*Spot medium*, *Spot large*, *On-demand medium*, *On-demand large*, *Local medium*⟩. The medium machines have 2 CPU cores while the large VMs have 4 cores. We use Amazon EC2's pricing policy on *us-east-1a* data center [6]. Each time-slot t in our model spans 10 minutes. The overheads for VM startup, checkpointing and migration between the cloud and local machine occupy one time-slot. On the other hand, the overhead for migration within the cloud needs zero time-slots, assuming it can interleave with VM startup time. For the job $J(C, D, T_{dl})$, we vary the compute requirement C between $2 - 8$ core-hours while the input data size and deadline constraint are also varied in 10 minute increments. Our focus is on *long-running* jobs (i.e., $O(hours)$) which cumulatively cost more and hence have more to gain.

The strategy table of RTBA algorithm is constructed on a server with 2x8-core AMD Opteron 3.0GHz CPU and 64GB RAM. The time to construct the strategy table for different C and T_{dl} values is measured. However, due to the limited space, we report the time to build the table for an 8 core-hour job with a 12 hour deadline as 24.74 minutes on average. For our SPSP model, $f_P^i\left(\cdot|P \neq a\right)$ can be obtained through

[6]Medium VM-hour costs $0.13, Large VM-hour costs $0.26, and the data transfer-in and -ot costs are $0.00 and $0.12 per GB

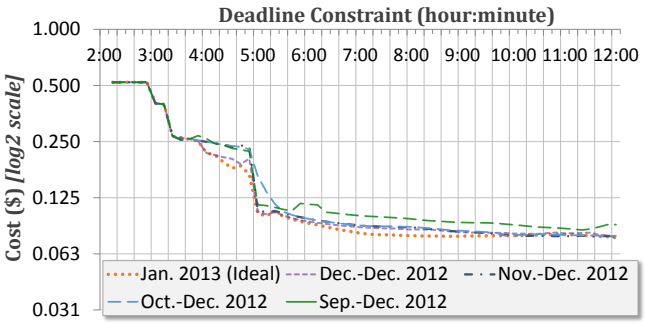


Figure 7: Average cost (Y-axis) to run 8 core-hour job in Jan, 2013 with various deadlines (X-axis) using RTBA. Plots show different training data used in the SPSP model. Using older data reduces cost benefits. Jan, 2013 line is ideal case.

the histogram of the spot price history. Since our SPSP model implies that the interval between two consecutive price changes forms a *geometric distribution* with parameter α_i, we set α_i as the inverse of the empirical mean of the intervals between two consecutive price changes [12]. We discretely choose the four most probable spot prices as our bidding prices, incremented by $0.001 as a safety buffer.

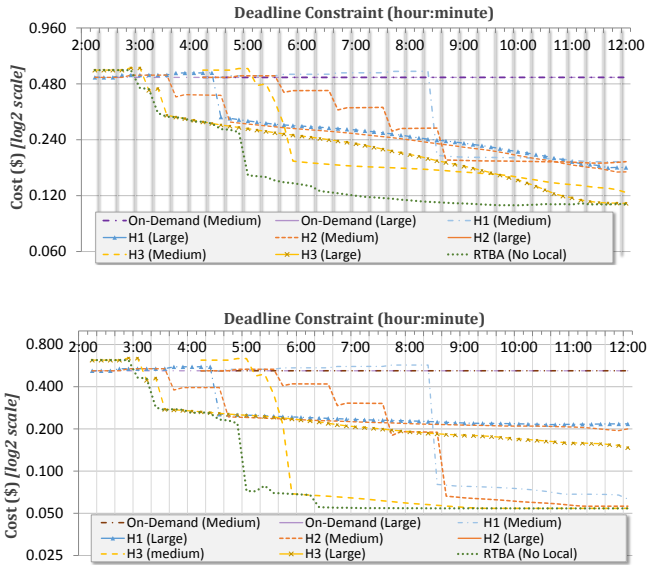
B. Impact of Training Interval

In the first experiment, the impact on cost reductions by using different training intervals to build the SPSP cost model is investigated. Four training interval durations in 2012, ⟨Sep-Dec, Oct-Dec, Nov-Dec, Dec-Dec⟩, are used to build four SPSP models, which are then used by the resource agent to construct a strategy table for scheduling the 8 core-hour job with varied deadlines in Jan, 2013. Further, we also build an SPSP model using Jan, 2013 data *post facto* as an ideal case. Fig. 7 reports the average of costs obtained for running the 8 core-hour job for each deadline, once for every 1 minute sliding window in Jan, 2013, i.e. each data point is averaged on $31 \times 24 \times 60$ runs.

When the training data is more outdated, the cost reduction becomes less. For e.g., in Fig. 7, *Dec-Dec, 2012* appears closest to the ideal case of *Jan, 2013*. This observation agrees with [2], which indicates that Amazon's pricing mechanism changes between *epochs*. Notably, while the SPSP model trained with older data digresses from the contemporary price pattern, the performance of RTBA only degrades slightly, demonstrating the robustness of our approach. So, it is possible to reuse, rather than often reconstruct, the strategy table even with varying price patterns.

C. Performance Comparison between Algorithms

In the second experiment, the performances of RTBA, H1, H2, and H3 are compared. Importantly, we investigate the influence of diverse job deadlines on the job's cost. As is intuitive, more relaxed deadlines offer more opportunities for the resource agent to help reduce costs. We use 8 core-hours as our candidate job's compute requirement.



(b) Simulation results for Dec, 2012

Figure 8: Average cost (Y-axis) to run 8 core-hour job with different deadlines (X-axis) using proposed strategies, in (a) Nov, 2012 and (b) Dec, 2012. The cost when using only on-demand medium or large VMs remains flat at $0.52.

We use spot prices from Aug-Oct 2012 as training data for the SPSP model. Two sets of simulations are conducted – for jobs submitted in Nov, 2012 and in Dec, 2012. The spot price patterns for these two months are very different (Fig. 1). The spot prices in Nov, 2012 have standard deviations of $\sigma_{medium} = \$4.88$ and $\sigma_{large} = \$0.1419$ for medium and large VMs. Nov is more variable than Dec, 2012 prices, which have $\sigma_{medium} = \$0.5284$ and $\sigma_{large} = \$0.3766$ for the respective VMs. In Fig. 8, we plot the average cost for completing the 8 core-hour job on the Y-axis with different deadlines on X-axis when using only on-demand VMs, or the RTBA, H1, H2, or H3 strategies. As before, we repeat the job for every 1 minute interval in Nov, 2012 (Fig. 8a) and Dec, 2012 (Fig. 8a), and report the average costs. Note that the on-demand approach and the heuristics operate on a single VM class during a job's life-cycle. While RTBA can switch between hybrid resources, we disable the access to local machine for a fairer, cloud-centric comparison. The heuristics use a bid price set at the most probable spot price, incremented by an additional $0.001 to offer a bid advantage. For H3, the effective frequency is statically set as the average lifetime of the considered VM class multiplied by its computing power (i.e., number of cores).

In general, except for the on-demand only approach, the costs reduce as the deadlines are relaxed for all our algorithms. RTBA is best able to exploit the looseness of the deadline and outperforms all others (green dotted line on the left), providing the minimum (optimal) costs. Further, we observe a few sharp drops, termed as *cost cliffs*, for our proposed strategies. These result from Amazon's policy of

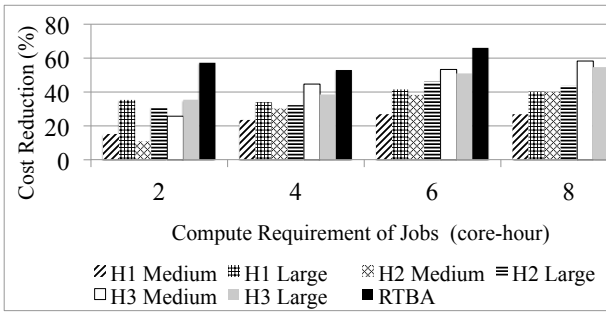


Figure 9: The cost reductions for the four algorithms rela to using only on-demand VMs are shown for Nov, 2 The heuristics H1–H3 only operate on a unique VM c while RTBA can switch VM classes. Hence, two colu (medium and large VM) are shown for each heuristic.

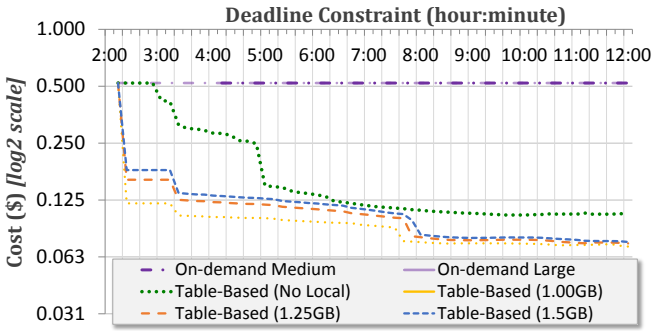


Figure 10: The costs (Y-axis) for processing 8 core-hour jobs with varied input data sizes. RTBA with and without the local machine converge as the deadline increases.

charging in hourly VM increments. Note that these cliffs are also visible as sharp color transitions in the heatmap (Fig. 5), that offers a parameter sweep of jobs and deadlines.

In RTBA, the cost cliffs appear earlier and are deeper than for H3, and similarly with H3 and H2, and with H2 and H1. More sophisticated approaches are more capable of advanced planning and checkpointing of the job's progress. The cliffs are inflection points. Deadlines that are in the proximity of these cost cliffs can benefit (or suffer) from small relaxations (tightening) of the deadlines it offer significant cost reductions (penalties). On the other hand, the heuristics also exhibit cases where they *overpay*, relative to the on-demand only approach. These result from the progress lost by the spot VMs due to out-of-bid events. In particular, H1 is the most vulnerable due to the absence of checkpointing while the overpayment is alleviated in H2 and H3. Notably, the performance of H3 is close to that of RTBA when the deadline constraint is loose, and the spot prices are stable, as shown at the tail of the curves in Fig. 8b.

D. Impact of Compute Requirement

In the third experiment, we use the same configuration as the second one except for varying the compute requirement for the job. The cost reductions achieved for our algorithms, normalized against using only on-demand VMs, are compared for a range of compute sizes (2, 4, 6, 8 core-hours), when run in Nov, 2012. The cost reduction for each compute size is averaged over a range of deadline constraints, from $1\times$ to $3\times$ of the compute size. As can be expected, Fig. 9 shows that the relative cost reduction of RTBA outperforms the other heuristics. However, as the compute requirement becomes larger, the advantage of RTBA shrinks. Further, the overhead to calculate the RTBA strategy table becomes acute for job's with large compute sizes. Given these, the RTBA and H3 algorithms are complementary; RTBA is advantageous for small sized or repetitive jobs while H3 is more useful for *ad hoc* job workloads or large jobs with punitive costs for building the strategy table.

E. Impact of Input Data Size

We investigate the relationship between a job's input data size D and cost to run the job. We retain the 8 core-hour job with different deadlines, and run it for Nov, 2012, averaging the costs from over a 1-minute sliding window. We compare RTBA with the on-demand only VMs, and further compare RTBA with and without access to local machines. Since the local machine is "free", there is an inclination to switch to them. However, as the job and its input data are initially in the cloud, there are bandwidth charges proportional to D that are incurred to ship the job to the local machine.

As is shown in Fig. 10, for the tightest deadline constraint, the four curves for the RTBA algorithms cost the same as using only on-demand VMs since there is not enough time to migrate out of the cloud or to use spot VMs. But, as the deadlines are relaxed, the difference between RTBA with and without the local machine narrows as the cost of getting the work done on cheaper spot VMs approaches the data transfer cost to the free, but scarce, local machine.

VIII. Conclusion and Future Work

Our work addresses the issue of effective and economic use of hybrid clouds for managing the life-cycle of long running jobs. It actively leverages the cheaply-priced spot VMs that have been less studied so far. A resource agent serves as a "personal" adviser for individual jobs running on exclusive VMs, with the goal of suggesting the right actions to meet the specified deadlines while minimizing rental costs. Our system and job models attempt to mimic real-world clouds and applications. Our simulation study is based on observed spot and on-demand VM prices on Amazon EC2, and incorporates both compute costs and asymmetric network bandwidth rates charged by cloud providers.

While the computational complexity of RTBA makes it tractable only for repetitive jobs, the H3 heuristic is simpler while nearing optimality in empirical studies. Hence, our work goes beyond a theoretical stochastic optimization problem and can be used in practice for average cost savings of $60\% - 72\%$ for H3 and RTBA.

Our approach can be extended along two dimensions. RTBA can be modified to accommodate the scenario where jobs are allowed to spillover the deadline but incur some penalty. The resource agent tends to be more aggressive when the penalty is outweighed by the saving which results from applying speculative strategies. Secondly, the QoS can be defined in terms of the rate of successful job completions, allowing the agent to balance risks and rewards.

The agent's role can be viewed from another perspective. As a *resource broker*, the agent owns some local resources to process end-user requests. By monitoring the spot market, the agent can decide whether to use its own resources or cheaper spot VMs to process jobs so that fragmented computing resources can be leveraged for productive work. This, in the spirit of the spot market, is a win-win-win situation for the broker, end-users and the cloud provider.

ACKNOWLEDGMENT

The authors would like to thank Viktor Prasanna, Alok Kumbhare, and Charith Wickramaarachchi of USC for their feedback. This research is supported by DARPA's XDATA program and National Science Foundation under award Nos. CCF-1048311 and ACI-1216898.

REFERENCES

[1] Amazon Spot Market. aws.amazon.com/ec2/spot-instances.

[2] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 Spot Instance Pricing. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec. 2011.

[3] A. Andrzejak, D. Kondo, and S. Yi. Decision Model for Cloud Computing under SLA Constraints. In *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, Aug. 2010.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *ACM Communications*, 53(4):50–58, Apr. 2010.

[5] D. P. Bertsekas. Dynamic Programming and Optimal Control. *Athena Scientific*, I, 2005.

[6] T. Bicer, D. Chiu, and G. Agrawal. Time and Cost Sensitive Data-Intensive Computing on Hybrid Clouds. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2012.

[7] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing Strategies for Parallel Jobs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[8] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See Spot Run: Using Spot Instances for Mapreduce Workflows. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, Jun. 2010.

[9] M. D. de Assuncao, A. di Costanzo, and R. Buyya. Evaluating the Cost-benefit of Using Cloud Computing to Extend the Capacity of Clusters. In *ACM International Symposium on High Performance Distributed Computing (HPDC)*, Jun. 2009.

[10] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The Cost of Doing Science on the Cloud: the Montage Example. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2008.

[11] S. Di and C.-L. Wang. Error-Tolerant Resource Allocation and Payment Minimization for Cloud System. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1097–1106, Jun. 2013.

[12] G. Grimmett and D. Welsh. Probability: An Introduction. *Oxford Science Publications*, 1986.

[13] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Data Sharing Options for Scientific Workflows on Amazon EC2. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2010.

[14] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. PaÃŰun, and S. Scott. An Optimal Checkpoint/Restart Model for A Large Scale High Performance Computing System. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2008.

[15] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Cost- and Deadline-constrained Provisioning for Scientific Workflow Ensembles in IaaS Clouds. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2012.

[16] M. Mao and M. Humphrey. Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[17] P. Marshall, K. Keahey, and T. Freeman. Elastic Site: Using Clouds to Elastically Extend Site Resources. In *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, May 2010.

[18] M. Mazzucco and M. Dumas. Achieving Performance and Availability Guarantees with Spot Instances. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*, Sep. 2011.

[19] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of A Scalable Multi-level Checkpointing System. In *ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2010.

[20] W. Voorsluys and R. Buyya. Reliable Provisioning of Spot Instances for Compute-intensive Applications. In *IEEE International Conference on Advanced Information Networking and Applications (AINA)*, Mar. 2012.

[21] S. Yi, D. Kondo, and A. Andrzejak. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. In *IEEE International Conference on Cloud Computing (CLOUD)*, Jul. 2010.

[22] M. Zafer, Y. Song, and K.-W. Lee. Optimal Bids for Spot VMs in A Cloud for Deadline Constrained Jobs. In *IEEE International Conference on Cloud Computing (CLOUD)*, Jun. 2012.

[23] H. Zhao, M. Pan, X. Liu, X. Li, and Y. Fang. Optimal Resource Rental Planning for Elastic Applications in Cloud Market. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2012.

[24] L. Zhao, Y. Ren, and K. Sakurai. A Resource Minimizing Scheduling Algorithm with Ensuring the Deadline and Reliability in Heterogeneous Systems. In *IEEE International Conference on Advanced Information Networking and Applications (AINA)*, Mar. 2011.

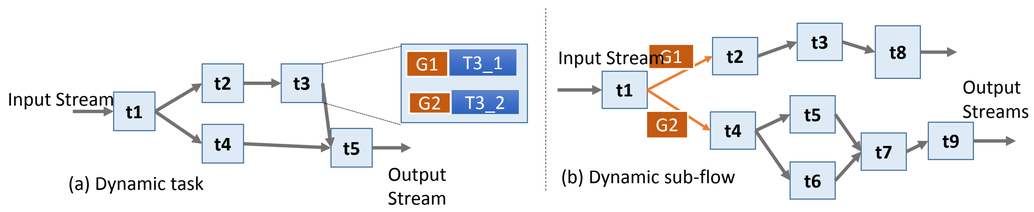
Constraint-Driven Adaptive Scheduling for Dynamic Dataflows on Elastic Clouds

USC Viterbi School of Engineering

Alok Gautam Kumbhare, 4th Year Ph.D. Candidate (kumbhare@usc.eu),
Viktor K. Prasanna (advisor), Yogesh Simmhan (advisor)

This work is supported by Defense Advanced Research Projects Agency's XDATA program under award No FA8750-12-2-0319 The views of authors expressed herein do not necessarily reflect those of the sponsors.

1. Dynamic Continuous Dataflows and QoS Metrics - Background



Dynamic Dataflow:

- Is a dataflow consisting of a number of ***dynamic tasks (or paths)*** connected using ***dataflow edge dependencies*** that continuously processing the incoming data streams and produces.
- It allows ***flexible re-configuration*** by allowing dynamic updates to the individual task logic or by updating an executing sub-flow.
- Extremely ***high and variable data rates***

Dynamic Tasks:

- A set of ***alternate implementations*** (alternates) for a given Logical activity.
- Each alternate exhibits different *"Cost"* to *"Value"* ratio.
- ***Cost*** for each alternate is measured in terms of *"Core-Seconds"* required to perform one ***Unit*** operation on a ***reference*** CPU core.
- ***Value*** is measured ***relatively*** among the alternates of a given Dynamic Task based on a domain specific ***utility function (****Domain Perceived Utility, e.g. Accuracy of Results)*

Application QoS:

- *Relative Application Throughput (Ω_i):* Ratio of *observed application throughput* to *maximum achievable throughput* given data rate r_t
- *Overall Application Value:* Alternate values *aggregated* for the application based on the alternate *active* during that interval

$$\Omega_i(t) = \frac{o_i(t)}{o_i^{max}(t)} \qquad \Gamma_i(t) = \sum_{p_i^j \in P_i} \left(\gamma_i^j \cdot A_i^j(t)\right)$$

2. Cloud Infrastructure modelling and characteristics

Cloud Modelling

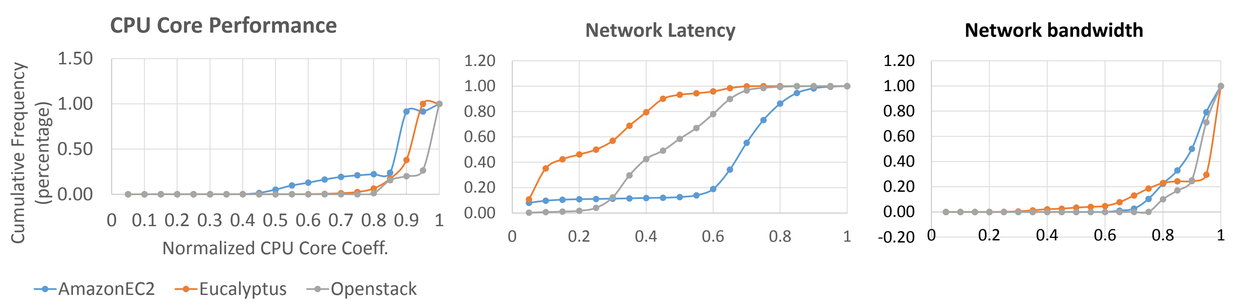
- Pay-per-use Model.
- ***Heterogeneous collection*** of unbounded on-demand resources.
- Typically, ***discretized cost model*** (e.g. hourly)
- Necessitates careful ***utilization of resources*** to ***minimize Monetory cost***, especially for long running continuous dataflows.

$$\mu_i[t] = \lceil min(t_{off}, t) - t_{start})/60 \rceil$$

- Susceptible to performance variations

Infrastructure Variability:

- ***Performance variations*** due to multi-tenancy, hardware generations etc.
- ***Temporal and spatial variations*** in performance.
- ***Predictable performance*** degradations: Usage cycles, Collocation (rack level, switch level, data center level etc.)

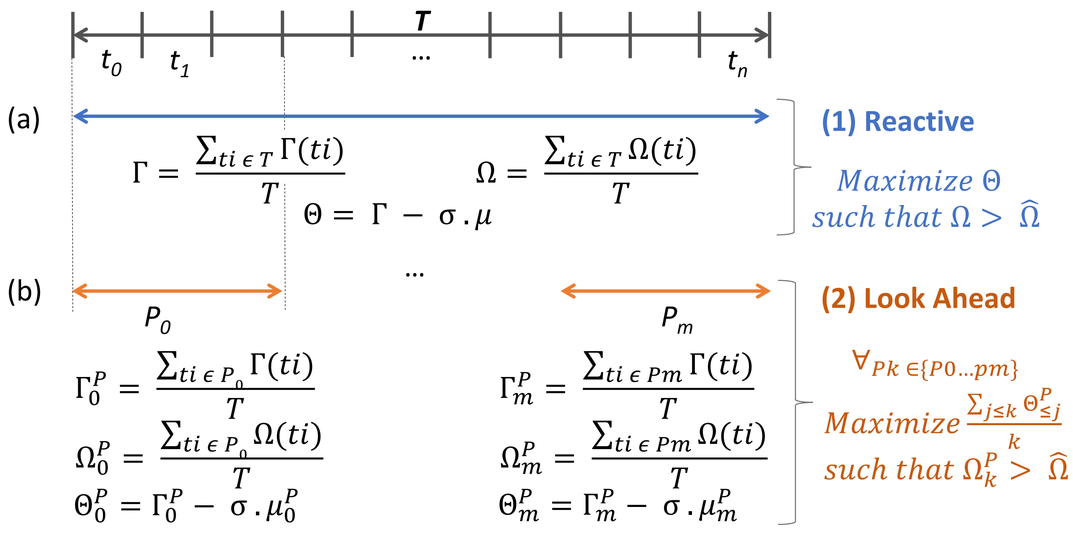


3. Constraint-Driven Adaptive Scheduling (Problem Statement)

Given: Optimization Interval, *T*. Monitoring interval *t*.

Goal : Select ***Active Alternates*** and ***Resource Mapping*** during each ***interval t*** so as to ***Maximize*** overall profit ***Θ*** over ***T***

Constraint: Average relative throughput $\Omega > \hat{\Omega}$

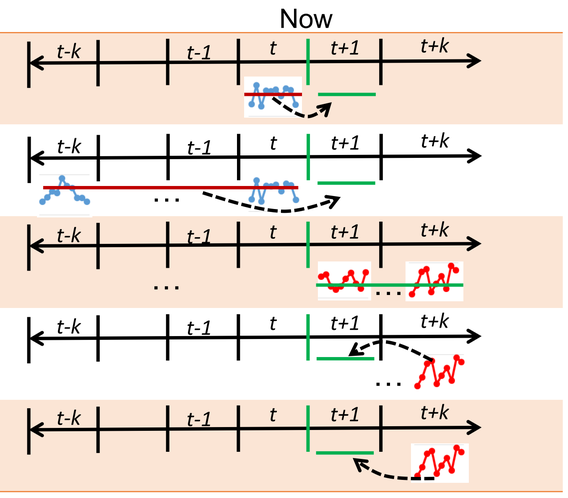


4. Reactive Scheduling Strategies (Proposed Solutions)

Two Step Process:

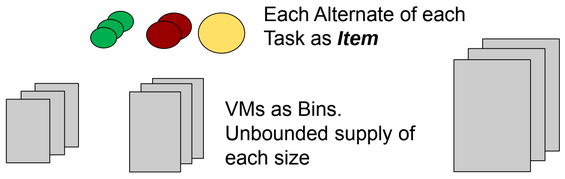
1. Get ***current "representative"*** resource performance and data rates ***for next interval***
2. Given the representatives, find the "best" resource mapping for the next interval

Representative Selection Heuristics

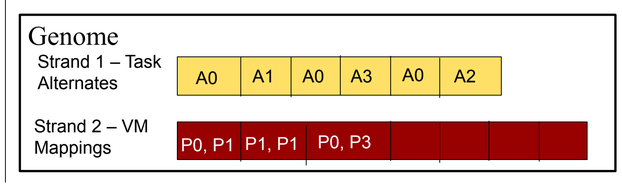


Resource Mapping Heuristics

Variable Sized Bin-Packing



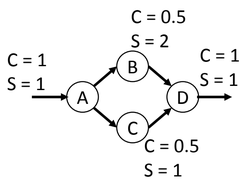
Genetic Algorithm



5. Predictive Look-Ahead Scheduling (PLAStiCC)

Main Idea: Find the Largest Interval where the current resource mapping Violates the constraint, create a **continuously refined** look-ahead plan with **Late Execution** model

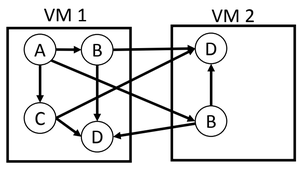
Example



(a) Sample Dataflow

Prediction Interval	data rate	vm1	vm2	Ω
t + 0	1.5	0.85	0.88	0.57
t + 1	2	0.92	0.94	0.46
t + 2	2	0.92	0.95	0.46
t + 3	1.2	0.91	0.92	0.76
t + 4	2.5	0.85	0.87	0.34
t + 5	2.2	0.86	0.89	0.43

(b) Data Rate and VM Performance Predictions

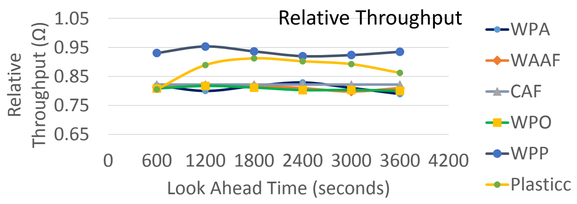


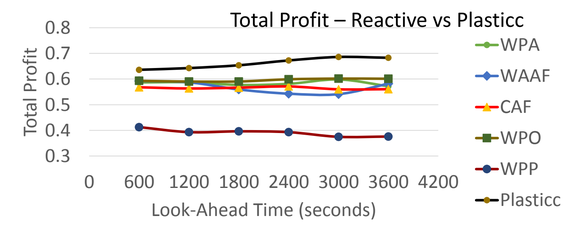
(c) PE Resource Mapping at time ***t***

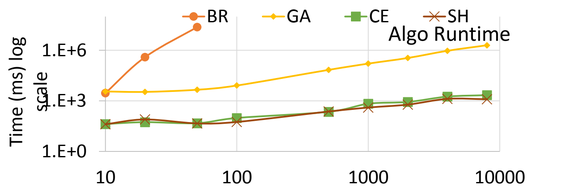
ot	ft						Planned Action
	0	1	2	3	4	5	
0	0.57	0.51	0.49	0.54	0.48	0.47	***Allocate*** new instance of A on VM 2 at ***t + 0***
0	0.77	0.68	0.66	0.73	0.65	0.64	***Allocate*** new instance of D on VM 2 at ***t + 0***
0							***Launch VM*** 3 at ***t + 3***
				0.84	0.79	0.77	***Allocate*** new instance of B on VM 3 at ***t + 3***
3						**0.88**	Done

(d) **PLAStiCC** iterations, Planned Actions and Cumulative **Ω**

6. Reactive VS PLAStiCC VS Genetic Algorithm – Observed Results







Conclusions:

- **Dynamic Dataflows provide powerful abstractions and flexible runtime adaptation.**
- **Reactive Scheduling Techniques address Infrastructure Performance and Data Rate variations. But are prone to trashing**
- **Predictive Look-Ahead Scheduling generates an evolving plan and mitigates trashing.**
- **Genetic Algorithms provide near optimal solutions but have high run-time overheads, and hence are suitable only for initial deployment**

PLAStiCC: Predictive Look-Ahead Scheduling for Continuous dataflows on Clouds

Alok Kumbhare[1], Yogesh Simmhan[2] and Viktor K. Prasanna[1]
[1]*University of Southern California, Los Angeles, California 90089*
[2]*Indian Institute of Science, Bangalore 560012*
Email: kumbhare@usc.edu, simmhan@serc.iisc.in, prasanna@usc.edu

***Abstract*—Scalable stream processing and continuous dataflow systems are gaining traction with the rise of big data due to the need for processing high velocity data in near real time. Unlike batch processing systems such as MapReduce and workflows, static scheduling strategies fall short for continuous dataflows due to the variations in the input data rates and the need for sustained throughput. The elastic resource provisioning of cloud infrastructure is valuable to meet the changing resource needs of such continuous applications. However, multi-tenant cloud resources introduce yet another dimension of performance variability that impacts the application's throughput. In this paper we propose *PLAStiCC*, an adaptive scheduling algorithm that balances resource cost and application throughput using a prediction-based look-ahead approach. It not only addresses variations in the input data rates but also the underlying cloud infrastructure. In addition, we also propose several simpler static scheduling heuristics that operate in the absence of accurate performance prediction model. These static and adaptive heuristics are evaluated through extensive simulations using performance traces obtained from public and private IaaS clouds. Our results show an improvement of upto 20% in the overall profit as compared to the reactive adaptation algorithm.**

***Keywords*-Continuous Dataflows; Predictive scheduling; IaaS Clouds; Elastic resource management; Stream processing**

I. Introduction

There has been a tremendous rise in the capability to collect data – pervasively and continuously – for a wide range of application domains such as financial trading, social networks, and cyber-physical systems (CPS) like Smart Power Grids and Smart Transportation. Consequently, there is increasing attention on data analytics frameworks that perform continuous data processing with low latency and guaranteed throughput. Batch processing systems [1], [2] that deal with high volumes of slow changing data abound. In contrast, continuous applications are characterized by data streaming with high velocity and at variable rates, on which multiple stages of information integration and analytics are performed. They offer online insight to domain experts (or their digital agents) to help detect and prevent frauds, power outages or security situations, as may be the case. Runtime environments for such applications demand *continuous adaptation of user logic and dynamic scaling of resources, under changing stream workloads*, to meet the user's quality of service (QoS) requirements.

Cloud computing platforms, especially Infrastructure-as-a-Service (IaaS), provide flexible, on-demand resources. Their elastic resource provisioning is well suited to scale resources at runtime in response to changing needs of continuous applications. Several stream processing frameworks [3], [4], [5] and adaptive provisioning algorithms [6], including our own earlier work [7], [8], utilize this online elasticity to balance resource cost against application performance in response to changing data rates.

However, cloud infrastructure exhibit *performance variability* for virtualized resources (e.g. CPU, disk, network) as well as services (e.g. NoSQL store, message queues) both over time and space. These variations may be caused by factors including shared resources and multi-tenancy, changing workloads in the data center, as well as diversity and placement of commodity hardware [9]. For time sensitive continuous dataflow applications, it becomes imperative to address such fluctuations in the performance of virtualized Cloud resources to ensure that the desired QoS is maintained. Strategies may range from simple over-provisioning and replication to dynamic application re-composition [8] and preemptive migration [10].

A pro-active management of application and resource mapping is possible with predictive models that can forecast resource behavior. Several performance models have been proposed for computational Grid infrastructure that also exhibit performance variability [11]. These models use the network topology and workload characterization to estimate the behavior of specific Grid resources [12], [13]. Similar models have also been developed for short and medium term predictions of performance metrics for virtual machine (VM) clusters using machine learning techniques such as discrete-time Markov chain [14]. Hence, it is feasible to ascertain the future performance of Cloud resources, to different degrees of confidence.

Given the ability to elastically control Cloud resources and the existence of resource performance prediction models, the question arises: *How can we replan the allocation of elastic resources for dynamic, continuous applications, at runtime, to mitigate the impact of resource and workload variability on the QoS of the application?*

In answering this question, we consider continuous dataflows as the abstraction used to compose these continuous applications. In our previous work [8], we considered

proposed adaptive scheduling heuristics for online task recomposition of dynamic and continuous dataflows as well as allocation and scaling of cloud resources. However, these strategies *reacted* to changes in workloads and resource variations to ensure QoS targets were met. In this paper, we leverage the knowledge of future resource and workload behavior to *pro-actively* plan resource allocation, and thus limit costly and frequent runtime adaptation. We propose a new Predictive Look-Ahead Scheduling algorithm for Continuous dataflows on Clouds (*PLAStiCC*) that uses short term workload and infrastructure performance predictions to actively manage resource mapping for continuous dataflows to meet QoS goals while limiting resource costs. We also offer, as alternatives, simpler scheduling heuristics that enhance the reactive strategy from our prior work. The specific contributions of the paper are as follows:

1) We analyze performance traces for public and private Cloud VMs to highlight spatio-temporal variations (§II). These motivate our problem while offering insights to build resource forecasting models; however, the modeling itself is beyond the scope of this paper.
2) We build upon our prior continuous dataflow abstraction (§III) to formalize their scheduling on Clouds VMs as an optimization problem (§IV), the solution for which will give an optimal mapping of tasks to cloud resources.
3) We propose several scheduling heuristics for elastic resources, including the novel PLAStiCC algorithm, that leverage short and medium term performance predictions (§V).
4) Finally, we evaluate the proposed heuristics through extensive simulations that use real performance traces from public and private Cloud VMs for several continuous dataflow applications and workloads (VI).

II. Analysis of Cloud Performance Variability

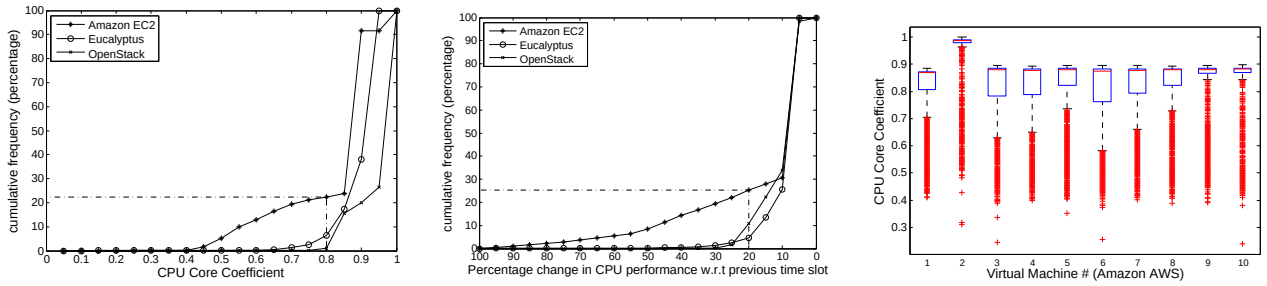
Performance variations present in virtualized and multi-tenant infrastructures like Clouds make it challenging to run time-sensitive applications such as continuous dataflows. Several studies have observed such variations both within a VM and across VMs [9]. Their cause is attributed to factors including multi-tenancy, evolving hardware generations in the data center, placement of VMs on the physical infrastructure, hardware and software maintenance cycles, and so on. In this section, further empirical evidence of such performance variation, both on public and on private clouds, are offered. This analysis motivates the need for our proposed predictive look-ahead scheduling algorithm and while offering insights on building performance prediction models used by our algorithm. Developing the actual performance prediction models are beyond the scope of this paper.

Two Cloud providers are considered: *FutureGrid*, a private academic IaaS Cloud running *Eucalyptus 2.0* and *OpenStack Grizzly (v7.0)* Cloud fabrics, supporting over 150 Cloud research projects [1], and *Amazon EC2*, a popular public commercial IaaS Cloud. The experimental setup consists of running three groups of VMs: 20 Eucalyptus, and 10 OpenStack VMs on FutureGrid, and 10 EC2 VMs on Amazon, over 1–26 day period. Each VM's class was equivalent to Amazon's `m1.small`, with one dedicated virtual CPU core and 1024 MB of memory. Standard CPU (whetstone[15]), disk, and memory benchmarks are run on each VM every 2 mins, and a peer-to-peer network benchmark between the VMs measure TCP connection time, packet latency and bandwidth ([16]). Here, we focus only on the CPU and network performance metrics since our proposed algorithms only account for these. However, this can be extended to consider variations in metrics like local disk I/O, durable storage and messaging services (Amazon S3, SQS), etc. based on the application's characteristics.

We are interested in the deviations of the VMs from their *rated* (i.e. ideal) performance rather than their absolute performance. Hence, for each of the three VM groups, each observed measurement for a performance metric is normalized based on the best observed performance for that metric, i.e., we divide each measurement by the best seen value in the VM group, which is the largest for bandwidth, and smallest time for CPU, network connect and latency. We also take the reciprocal for metrics where smaller is better so that in these plots, 1.0 always refers to the best seen value.

Figure 1(a) shows the cumulative distribution function (CDF) for the normalized CPU core performance (X axis) across all VMs and across time, in each of the three VM groups. The Y axis shows the cumulative fraction of the number of VMs whose observed performance is less than the normalized performance on the X axis. We see that all three VM groups and service providers show variations in performance, with the EC2 VMs on Amazon's public Cloud showing the highest variations, possibly due to greater data center diversity and high multi-tenancy and utilization of physical resources. We also notice that many VMs fail to reach the ideal 1.0 normalized performance. 23% of EC2 VMs have a normalized core performance of < 0.80, while $< 10\%$ of the FutureGrid VMs exhibit this shortfall, possibly due to more uniform physical resources. *A wide divergence in rated and available computational capability can tangibly impact the overall application QoS* (§ VI). Variations are also seen in the network performance measures when considering all VMs over time; we do not plot this in the interests of space. In brief, we see a high deviation in packet latency for Amazon EC2, with more than 45% of VMs exhibiting a normalized packet latency greater than 0.70; the network bandwidth for 22% of EC2 VMs have a bandwidth coefficient < 0.80). These can punitively impact the exchange of small messages by streaming applications. These performance variations in CPU and network characteristics

[1] https://portal.futuregrid.org/projects-statistics



(a) Cumulative Distribution Function (CDF) of normalized CPU Core performance, across VMs and over time.

(b) CDF of performance variation relative to previous measurement time-slot.

(c) Box-and-Whiskers plot of normalized performance distribution over time (Y axis) and for different Amazon AWS VMs (X axis).

Figure 1. CPU Core performance characteristics, across space and time.

cause the application's QoS to degrade and demonstrate the need for dynamic runtime adaptation.

Our prior work on dynamic application adaptations [8] accounted for the diversity in performance across VMs and small/slow changes in performance using reactive (after-the-fact) strategies. However, we also observe transient and sharp fluctuations in performance (i.e. high amplitude of Δ, small duration), both in CPU as well as network performance, within the same VM. Figure 1(b) shows the percentage change (unsigned) in average normalized CPU performance between consecutive 5 minutes measurement periods. Larger the % value of the X Axis (Left), greater the variation in performance between consecutive periods, and more pronounced is the short term performance fluctuations. In about 25% of the observations, the performance changes by $> 20\%$ within a 5 min period for an EC2 VM. The extent of variations is also different for different VMs. Figure 1(c) shows a box-and-whiskers plot on the Y axis representing the average normalized CPU performance for each 5 min interval and the X axis representing different EC2 VMs. The height of the box shows the difference in performance between q_1 and q_3 quartiles (i.e. 25^{th} and 75^{th} percentiles). Also, a number of outliers that fall below ($< q_1 - 1.5 \times (q_3 - q_1)$ are also observed. Figure 2 shows the temporal deviations between successive 5 min time slots for the same VM for the network characteristics. Here too we see that for $20 - 30\%$ of the cases, there is a $> 20\%$ change in network performance, depending on the metric. A detailed discussion is omitted for brevity.

Such performance fluctuations and outliers mean that the immediate past performance of a resource may not be sustained in the immediate future, and assuming so will cause sub-optimal resource allocation strategies. In particular, such transient performance changes can cause decisions that are unrolled soon after. A look ahead adaptation strategy that not only accounts for the performance variations across VMs, but is also resilient to temporal changes in the short and long terms within a VM is necessary.

III. Preliminaries & Background

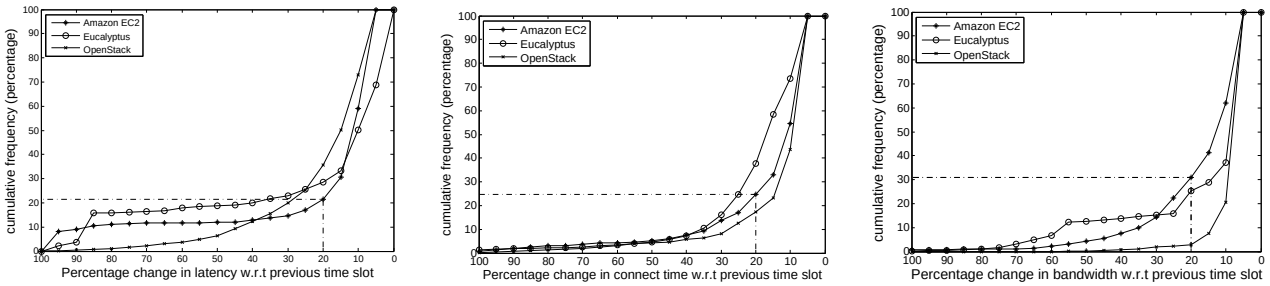
We introduce the abstraction of continuous and dynamic dataflows, and summarize our existing work on reactive scheduling of such dataflows on Clouds with infrastructure variability [8]. We build upon this for our current contribution on pro-active scheduling with look ahead.

A *continuous dataflow*, also known as a stream processing application, is a directed acyclic task graph (DAG), $G = (P, E, I, O)$, where $P = \{P_i\}$ is the set of *processing elements (PEs)*; $E = \{e_{i,j} : \exists$ a dataflow edge between P_i and $P_j\}$ is a set of dataflow *channels* between the PEs; and $I \neq \varnothing \subset P$ and $O \neq \varnothing \subset P$ are the sets of input PEs (without an incident channel) and output PEs (without an outgoing channel), respectively. Each PE is a long-running user-defined compute logic, accepting and processing data messages on the incoming channels and producing data messages on the outgoing channels. This DAG-based application composition model, prevalent in various domains, provides a flexible abstraction to build complex continuous dataflow applications [17], [18].

A. Dynamic Continuous Dataflows

Continuous dataflows are extended to dynamic dataflows that allow run-time application re-composition in response to the domain triggers. They also provide an additional dimension of control to schedule the dataflow to balance application QoS, resource cost and application value.

The main characteristic of a *dynamic dataflow* is the set of "alternates" [19] available for individual PEs. This lets the user define more than one implementation (alternates) for each PE. All alternates for a PE perform the same logical operation but with a different trade-off between the domain perceived "value" and the resource needs. Only one alternate is active for a PE, and the choice of activating an alternate does not impact the dataflow correctness, nor does changing the active alternates while the continuous dataflow is executing. The latter allows the framework to schedule any alternate for a PE without affecting application correctness, just quality. Each alternate p_i^j of a PE (P_i) is associated with the QoS, resource and execution metrics: relative domain value, γ_i^j, resource cost, c_i^j, and selectivity, s_j^i.



(a) Latency (b) Connect (c) Bandwidth

Figure 2. CDF of variation in normalized network performance measures relative to previous measurement time-slot.

The *relative domain value*, $0 < \left(\gamma_i^j = \frac{f(p_i^j)}{\max_j f(p_i^j)}\right) \leq 1$, is the domain perceived relative benefit of activating that alternate for a given PE relative to its best available alternate, where $f(p_i^j)$ is a user-defined positive valued function, e.g., the F_1 statistical measure. In a timestep, t, the *application value* for the entire dataflow is the aggregation of all active alternates' values, one for each PE, in that duration.

The *resource cost*, c_i^j, is the number of core-seconds required to process a single incoming message by an alternate on a "standard" CPU core. This restricts the PE's internal logic to use a single-core implementation and a *standard CPU core* is an infrastructure specific *benchmarked* (as opposed to rated) unit of computing in a heterogeneous Cloud environment. These allow us to use a simple linear scaling to obtain the processing requirements for that alternate on other benchmarked CPUs. For e.g., given an alternate that requires 3.6 secs to process a single message on a standard 1 Ghz core, the time required to process that alternate on another core with processing capacity benchmarked at 1.8 Ghz can be calculated as, $processing\_time_x = \frac{processing\_time_{std}}{processing\_capacity_x} = \frac{3.6}{1.8} = 2secs.$

Selectivity, s_i^j, is the ratio of the number of messages *produced* by the alternate to the number of messages *consumed* by it to complete a logical unit of operation by that alternate. Selectivity determines the message load on the downstream PEs in the dataflow. The impact of an alternate on overall dataflow cost is thus its own resource cost and its contribution to the cost of downstream PEs.

The dataflow application is deployed in a distributed environment [20]. One alternate is active for each PE at deployment and the scheduler can switch between alternates during the application's runtime. Several *instances* of an alternate can run in parallel across independent cores; while instances may inter-communicate through some persistence mechanism, for simplicity, we assume that each instance of a PE is data parallel and can execute independently. So, a PE can be scaled up/down by increasing/decreasing the number of instances, each using an *exclusive* core. [8] illustrates such dataflows, alternates, and their resource mapping to VMs.

The QoS of the application is measured by the *relative application throughput*, Ω, which is the ratio of observed output throughput to the maximum achievable output throughput, given the incoming data rate to the dataflow. The observed throughput depends on the number of instances of each PE and the resources assigned to them. The dataflow's lifetime is divided into *optimization periods* of duration T, and the user specifies as a QoS constraint the *minimum average* relative throughput, $\widehat{\Omega}$, to be met for each T.

The *overall profit*, Θ, for the application deployment is defined as $\Theta = \Gamma - \sigma \times \mu$, where $0 \leq \Gamma \leq 1$ is the normalized application value and μ is the dollar cost incurred by the application, both defined over each optimization period T, and σ is the *value coefficient* defined as:

$$\sigma = \frac{\text{Max Application Value} - \text{Min Application Value}}{\text{Acceptable Cost at MaxVal} - \text{Acceptable Cost at MinVal}}$$

The optimization goal for scheduling dynamic continuous dataflows, thus, is to maximize the overall profit, Θ, while meeting the user defined QoS constraint $\Omega \geq \widehat{\Omega}$ for each optimization period.

B. Reactive Runtime Scheduling and Resource Mapping

To achieve the optimization goal, the scheduler can, at runtime: (1) switch the alternate for a PE, (2) change the number of instances for an alternate, and (3) select the Cloud VMs on which instances are deployed. These can be in response (i.e. reactive) to changes in the incoming data rates and variability in infrastructure behavior (or even the scheduler's prior sub-optimal decision), that cause a deviation from the goal. Our prior work on a reactive scheduling algorithm [8], discussed in brief here, forms the basis for the proposed PLAStiCC algorithm (§ V).

The scheduling algorithm has two phases [8]. In the *initial deployment phase*, the algorithm assumes that the rated performance for the Cloud resources and estimated data rate suggested by the user are accurate, and allocates resources to each PE. This determines the initial alternates that are active, the number of data parallel instances of each, the number and size of VMs to be launched, and the mapping of PE instances to these VMs. In the *runtime adaptation phase*, the scheduler continuously monitors the dataflow and infrastructure, measuring the current relative throughput, incoming data rates, and the CPU performance

for, and bandwidth between, different VMs. Based on these *observed* performance metrics, it can decide to change the active alternates, and scale up/down the number of VMs and PE instances to balance the QoS against resource cost.

The reactive algorithm [8] has two independent stages, *alternate re-deployment* and *resource re-deployment*. The former selects feasible alternates for each PE such that they do not require any changes to the resources required. This depends not only on the active alternates but also on the current resource performance. It then picks the alternate for a PE with the highest value relative to its cumulative cost, i.e., the sum of its own resource needs and resources needed by downstream PEs due to its selectivity. Resource re-deployment, on the other hand, preserves the active alternates while it scales up/down the number of instances of the alternates and launches/releases VMs. Here, the heuristic makes several decisions: *Which PE is under/over provisioned?*, *Which PE instance has to be scaled down?*, and *Which VM has to be shutdown?*. These decisions affect the overall application QoS, its value, as well as the resource cost. [8] offers a comparison of these heuristics for runtime adaptation.

The scheduler uses the monitored application and infrastructure metrics to decide on initiating alternate or resource re-deployment. The "reactiveness" is because the scheduler's decisions assume that the observed data rates and resource performance from the immediate past will sustain in the future. Although this reactive algorithm mitigates the effects of variations in workload and infrastructure performance, due to its myopic nature, it is susceptible to spikes and short term fluctuations in the VMs' performance. This leads to adaptation decisions (e.g. launching a new VM in response to a transient performance drop) that need to be reversed in the near future (e.g. once the VM's performance is regained). Besides adding to the scheduling overhead, it also increases the cost of Cloud resources that are billed by the hour.

IV. Problem Formulation

Reactive scheduling can be mapped to the constrained utility maximization problem applied to continuous dataflows [8]. We define the pro-active *look-ahead optimization* problem as a refinement of this by including predictions for the incoming data rates and performance of the underlying infrastructure. In contrast, the reactive problem only has access to the current snapshot of the behavior.

We define a fixed optimization period T over which the application schedule is to be optimized for while satisfying the QoS constraint. T may be the duration of the application lifetime or some large time duration. T is divided into a number of fixed length timesteps, $t_0, t_1, ..., t_n$. The initial deployment decision, based on the rated infrastructure performance and user estimated data rates, is done at the start of timestep t_0, the system is monitored during t_i and runtime decisions performed at the start of timestep t_{i+1}.

Figure 3 (a) shows the *reactive optimization problem* defined over an optimization period T. The profit (i.e. utility) function to maximize, Θ, is defined over the normalized application value and the total resource cost over the entire period T, given by $\Theta = \Gamma - \sigma \times \mu$. Similarly, the application throughput constraint $\Omega \geq \widehat{\Omega}$ is defined over the average throughput observed over T. Consequently, the instantaneous Ω at a given time may not satisfy the given constraint though the average is satisfied by the end of T.

This problem is modified for pro-active *look-ahead optimization* as follows. Since prediction algorithms are often accuracy over short horizons (i.e. near future), and to reduce the time complexity of the problem, we reduce the above global optimization to a set of smaller local optimization problems (Figure 3 (b)). First, the optimization period T is divided into *prediction intervals* $[P_0, ..., P_k]$, each of size equal to the length of the prediction horizon given by the prediction model ("oracle"). As before, we divide each interval P_j into timesteps $t_0^j, t_1^j, ..., t_k^j$, at the start of which the adaptation decisions are made.

Next, a stricter version of the application constraint is defined, requiring it be satisfied for each prediction interval P_j, i.e. $\forall P_j : \Omega_j^p \geq \widehat{\Omega}$. It is evident that if this constraint is satisfied for each interval, it will be satisfied over the optimization period; however, the converse may not hold.

Finally, in varying the profit function, Θ, we cannot define it for each interval since Θ is cumulative over the resource cost, and the VM provisioning may span multiple prediction intervals. Rather, we optimize the profit calculated "so far", given as the cumulative average over each of the previous intervals, i.e., $\Theta^p_{\leq j} = \frac{\forall_{i \leq j} \sum \Gamma_i^p}{\sum P_i} - \mu_{\leq j}$, where $\mu_{\leq j}$ is the cumulative cost till prediction interval P_j. Further, while conciseness precludes a formal proof, it can be shown by contradiction that the global profit function Θ defined earlier is equivalent to $\Theta^p_{\leq k}$, where P_k is the last prediction interval in the given optimization period T.

The look-ahead optimization problem described above assumes a perfect "oracle" prediction model that can see accurately into the entire prediction interval. The predictions obtained at the start of each timestep t_i within the interval are used to *plan* adaptation actions (such as scale up/down instances) for each timestep in the rest of the interval. But these actions are *enacted* only when the timestep is reached.

With a perfect oracle, the predicted values for an interval remain fixed for that interval. However, given the limitations of state-of-the-art multivariate time series models [21], in practice, the errors in the predicted values will increase with the horizon of prediction. Hence, we will get better prediction values as we move further into a prediction interval. To utilize this, we allow new predictions to be made at each timestep t_i rather than at the end of each interval. Further, anytime a new (improved) prediction changes ex-

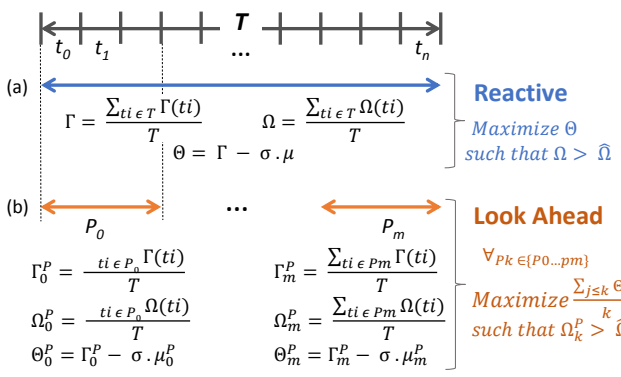


Figure 3. The (a) reactive and (b) look-ahead runtime optimization problems defined over the optimization period T. Reactive optimizes over globally T while look-ahead does it piecewise over each interval P_j. They observe/predict during timesteps t_i and take decisions at the start of t_{i+1}.

isting predicted values, we start a new interval and replan actions for all its timesteps, replacing the prior plan.

This sliding window of replanning does not change the optimization problem, except that the optimization decisions planed in one interval may be altered in the succeeding interval if updated predictions are available. However, note that this is different from the reversal of decisions in the reactive version, since the decisions there are enacted immediately for the next timestep whereas in the look-ahead version the replanning may just change a future action and hence not incur resource penalty.

V. Scheduling Heuristics

As with reactive scheduling (§ III-B), the look-ahead scheduling consists of two phases, the *initial deployment phase* which maps the application to VMs in the Cloud assuming rated performance and that the estimated incoming data rate holds; and the *runtime adaptation phase* which alters the resource allocation and alternates to account for the observed and predicted behavior. In this paper, we retain our earlier initial deployment algorithm [8] and focus on the latter phase. As such, the impact of the initial deployment is amortized by the runtime adaptation for long running dataflows. We first propose a look-ahead scheduling algorithm, *PLAStiCC*, which uses the predicted resource performance and data rates to generate dataflow adaptation plan for the near future. In addition, we propose several simpler heuristics that use averaging models to estimate performance and data rates. These, while out-performing reactive scheduling in many cases (§VI), offer a different trade-off between profit and scheduling cost than PLAStiCC.

A. Predictive Look-Ahead Scheduling (PLAStiCC)

The PLAStiCC algorithm is run at the start of each prediction interval. It plans a set of adaptation actions for timesteps in that interval to maximize the cumulative profit while meeting the constraint for that interval. The key intuition is to identify the farthest timestep in the current interval for which the existing deployment meets the QoS constraint for messages expected in that interval, given the performance and data rate predictions. The *cross-over* timestep indicates the future timestep beyond which the current deployment fails to meet the QoS. Once identified, we replan the deployment starting at the cross-over timestep until the last timestep of the interval, recursively. The result of this algorithm is zero or more cross-over timesteps, identified in the interval, where the QoS constraint fails and the planned adaption action to be taken at each timestep.

Algorithm 1 shows the PLAStiCC runtime adaptation algorithm. It takes as input the continuous dataflow DAG D, the predicted data rate $PR^i_{t..t+n}$ for each input PE P_i as a time series $[t..t+n]$ for timesteps in the interval, and the predicted VM performance time series $PV^j_{t..t+n}$ for each VM_i, which includes CPU and inter-VM network behavior. The algorithm outputs a plan of actions X to be taken at timesteps in the prediction interval $[t, t+n]$.

PLAStiCC is called as and when new predictions are made. It is run before the first timestep, t, of the prediction interval and tries to identify the farthest *non*-cross-over timestep. The algorithm maintains two pointers, ot, which is the last-known cross-over timestep in the interval, initially blank, and ft, the last timestep of the interval, set to $ft = t + n$. It tries to locate the last timestep when the aggregate constraint is met (lines 5–13). Starting at the last timestep, it calculates the aggregated throughput (Ω) by computing the processing capacity available to each PE (*Alloc*), the cumulative incoming data messages at each PE using the predicted data rates and the network behavior (*M*), and the aggregated relative throughput resulting from these values. If this aggregate throughput is not within $\hat{\Omega} \pm \epsilon$, we try the previous timestep (line 12). If it is met, have found the cross-over boundary between constraint being satisfied and not (line 10). If the constraint is met in the first iteration, i.e. at the interval's end, no adaptation is needed (line 15).

Once the cross-over is identified, ot is set to this last-known cross-over timestep, and ft reset to the interval end $t + n$ for the next iteration. We then decide the action to be taken at the cross-over to meet the constraint. For this replanning, we get the aggregated values for the available resources, the incoming messages and the relative throughput for the rest of the interval after the cross-over, $(ot, t+n]$ (lines 18–20). Since the expected throughput is deviating for this period, we either *scale up* ($\Omega_{ot..t+n} < \hat{\Omega} - \epsilon$) or *scale down* ($\Omega_{ot..t+n} > \hat{\Omega} + \epsilon$)), *incrementally* (lines 21–27).

Both the ScaleUp and the ScaleDown functions try to balance the overall application value Γ (Sec. III-A) and the total application cost μ. The ScaleUp function first identifies the "bottleneck" PE with minimum relative throughput, lying on the critical path from the input to output PEs [8]. It can then either switch to an alternate for this PE with lower processing needs (and potentially decrease the

application value) or increase the number of instances (and possibly increase the resource cost) for the bottleneck PE. This choice is based on the local optimization of the profit function Θ_t which uses the user specified σ coefficient that determines the "acceptable" value:cost ratio.

Specifically, the cost of increasing an instance is zero if a VM has spare cores or else is equal to the cost of a new VM. Then, the change in application value from switching to the alternate with next lower processing needs is calculated. The change in Θ_{ot-ft} from either of these actions is calculated, and the adaptation that results in higher profit is selected. Similarly, the SCALEDOWN function identifies an over-provisioned PE and makes a choice between switching to an alternate with higher value or decreasing the number of instances, whichever leads to higher profit.

Once such incremental action is planned for the cross-over timestep, we resume evaluating the planned deployment ensure it satisfies the constraint for the rest of the interval, $(ot, t+n]$ (lines 5 - 13). The replanning is incrementally done till all cross-over timesteps and adaptations are identified for the interval, and the resulting plan satisfies the constraint for the entire interval. Finally, the algorithm performs a repacking of VMs to collocate distributed PEs as well as shuts down those that are unused (line 30).

B. Averaging Models with Reactive Scheduling

The PLAStiCC algorithm reduces to the reactive algorithm if the length of the prediction interval is exactly one timestep, i.e. only current and no future values for data rates and infrastructure performance are given. Rather than use fine-grained predictions (PLAStiCC) or assume that the most recent observed values will sustain (reactive), we develop several heuristics that use simple but intuitive estimates of future behavior based on aggregated past behavior. This also addresses PLAStiCC's susceptibility to prediction errors since it overfits the schedule to the predictions. Similar to the reactive scheduler, these heuristics are executed only when the application QoS deviates from the constraint. The heuristics each differ in the type of averaging done over observed values to estimate future behavior of resource performance and data rates.

1) *Current After-the-fact (CAF)* This chooses observed value in the most recent timestep as the estimated future behavior for all timesteps. This is same as the classic reactive scheduler [8]. This reacts after-the-fact to the observed resource and data rate changes.
2) *Windowed Average After-the-fact (WAAF)* This uses the average seen over several recent timesteps as the future estimated behavior. This smooths out spikes while accounting for the recently observed trends that might last in the near future.
3) *Windowed Predictive Average (WPA)* This averages the values predicted for all timesteps by the advanced prediction model used by PLAStiCC, and uses the single average value in all timesteps in the interval. This mitigates the impact of large prediction errors in a few timesteps, where the PLAStiCC is susceptible.
4) *Windowed Predictive Optimistic (WPO)* This too uses the advanced prediction model of PLAStiCC but picks the most favorable infrastructure performance (highest) and data rate (lowest) from among all timesteps predicted for, and uses that single value for all timesteps. This offsets the impact of models that under-predict [22].
5) *Windowed Predictive Pessimistic (WPP)* This is similar to WPO, except it picks the least favorable performance and data rate prediction as the single value. This handles models with over-prediction bias [22].

Algorithm 1 PLAStiCC Runtime Adaptation Heuristic

```
1: procedure PLASTICC(Dataflow D, MinThroughput Ω̂_t,
   DataRatePredictions PR_{t..t+n}, VMPerfPredictions PV_{t..t+n})
2:     ot ← ∅, X ← ∅                          ▷ Init cross-over list to null
3:     while ot ≤ t+n do
4:         ft ← t+n
5:         while ft ≥ t do                    ▷ Work back from last timestep.
6:             Alloc ← AVAILABLERESOURCEALLOC(PV_{ot..ft})
7:             M ← CUMULATIVEINCOMINGMSGS(PR_{ot..ft})
8:             Ω ← CUMULATIVETHROUGHPUT(D, M, Alloc)
9:             if Ω̂ − ϵ < Ω < Ω̂ + ϵ then
10:                Break                      ▷ QoS constraint met at this timestep.
11:            end if
12:            ft ← ft - 1                    ▷ QoS not met. Work backwards.
13:        end while
14:        if ft = t+n then                   ▷ QoS met at end of interval.
15:            Break                          ▷ No more replanning. Done.
16:        end if
       ▷ Replan actions for the remaining duration (ft, t+n]
17:        ot ← ft                            ▷ Update last-known cross-over.
18:        Alloc ← AVAILABLERESOURCEALLOC(PV_{ot..t+n})
19:        M ← CUMULATIVEINCOMINGMSGS(PR_{ot..t+n})
20:        Ω ← CUMULATIVETHROUGHPUT(D, M, Alloc)
21:        if Ω < Ω̂ then
22:            bpe ← BOTTLENECK(D,CA)
23:            X ← SCALEUPORDECAPPVALUE(bpe, ot)
24:        else if Ω > Ω̂ then
25:            ope ← OVERPROVISIONED(D,CA)
26:            X ← SCALEDOWNORINCAPPVALUE(bpe, ot)
27:        end if
28:    end while
29:    X ← REPACKANDSHUTDOWNVMS( )
30:    return X   ▷ Return cross-over timesteps and action plan.
31: end procedure
```

VI. EVALUATION

Simulation Setup: We evaluate PLAStiCC and the different averaging models for reactive scheduling through a simulations study. We extend the CloudSim [23] data center simulator to *IaaSSim* [2], that additionally incorporates temporal and spatial performance variability using real traces

[2]http://github.com/usc-cloud/IaasSimulator

collected from IaaS Cloud VMs. Further, our *FloeSim* [3] simulates the execution of our Floe stream processing engine [20], on top of IaaSSim, with support for continuous dataflows, alternates, distributed deployment on VMs, dynamic instance scaling and a plugin for different schedulers.

We use Amazon EC2's performance traces (§ II) in IaaSSim. For each simulation experiment, we deploy a dataflow to the Cloud using FloeSim, run it for 12 hours (simulated) at a stretch – which is also the optimization period ($T = 12hrs$), and repeat it several times to get an average. We use a timestep duration of $t = 5mins$. Experiments vary the scheduler used, and, for PLAStiCC, the prediction interval changes between $P = 10 - 30mins$.

Synthetic Dataflows and Streams: We evaluate the adaptation algorithms using a synthetic dataflow in Figure 4 with 10 stages of 3 PEs each, and 3 input and output data streams. The dataflow is randomly perturbed to generate between 1–3 alternates each, with different cost, value and selectivity. This gives us a total of 30 PEs and ∼45 alternates.

Each of the three input data streams are independently generated based on observations in domains like Smart Power Grids [18]. Starting from a base stream with sinusoidal data rates whose peak and trough range from $75 - 2$ messages, and with wave length of $24hrs$, we add random noise every $1min$ that varies the rate by up to $\pm 20\%$.

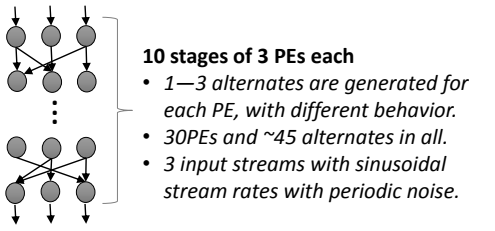


Figure 4. Sample dataflow pattern used for evaluation

Evaluation Metrics

Overall Application Relative Throughput (Ω):: This is the aggregate relative throughput obtained at the end of the optimization period. We check if the heuristic satisfies the QoS constraint $\Omega \geq \widehat{\Omega}$. A higher value of Ω beyond $\widehat{\Omega}$ is not necessarily good unless it also gives a higher overall profit.

Overall Application Profit (Θ): : This is the aggregate profit $\Theta = \Gamma - \sigma \times \mu$ obtained at the end of the optimization period. Given two algorithms which satisfy the QoS constraint, the one with a higher overall profit is better. Note that profit is *unitless* and can only be compared relatively.

Heuristic Stability (χ):: This is the frequency of deviation in instantaneous relative throughput Ω_t from the $\widehat{\Omega} \pm \epsilon$ during the optimization period. For reactive algorithms, this is the number of times adaptation actions respond to the deviation, while for PLAStiCC, this is the number of time the *planned* actions failed to meet the constraint. A lesser number indicates a more stable heuristic.

[3]http://github.com/usc-cloud/FloeSim

A. Experiments

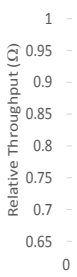
1) Scheduling under Ideal Predictions: We evaluate the proposed heuristics for a perfect oracle model where the predictions for resource performance and stream rates are accurate. Further, we evaluate them with prediction intervals ranging from 10 mins (i.e. two $5min$ timesteps per interval) upto to 30 mins and study the effect of the length of prediction interval under perfect oracle.

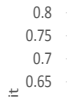
2) Scheduling under Predictions with Bounded Errors: We run experiments where the model predicted performance and data rate values include bounded errors, $\hat{e}$, to simulate short-comings of real world prediction models. The predictions have smaller error for near timesteps and larger ones for farther timesteps. First, we linearly scale the error from *zero* for the current timestep (e_0) to the $2 \times \hat{e}$ for the last timestep (e_n) in the prediction interval, to give a mean close to $\hat{e}$. Then, we sample the actual error for a timestep i from a normal distribution with mean 0 and variance equal to $e_i/2$, and use this as the actual error for that timestep to obtain the predicted value. We can also optionally specify a "bias" of $\pm 5\%$ that is added to get a positive or negative biased model. We make predictions every $10mins$ for all models.

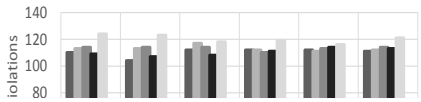
B. Results

Perfect Oracle Model: We first present the results obtained for the perfect oracle model, which signifies the best case results for the PLAStiCC heuristic as compared to the various reactive models. Figure 5 shows comparison between the PLAStiCC algorithm to the reactive models for different metrics (Ω, Θ, and χ) with different prediction interval lengths for which the oracle predictions are available). Figure 5(a) shows the overall relative throughput for various algorithms observed at the end of the optimization duration. We observe that all the adaptation algorithms successfully satisfy the application constraint ($\Omega \geq 0.8$) over the optimization duration for prediction interval lengths.

Higher values of relative throughput do not necessarily mean a better algorithm. We compare this with Θ, and χ. Figure 5(b) shows the overall profit ($\Theta = \Gamma - \sigma\mu$) obtained for different algorithms. First we observe that the PLAStiCC algorithm performs consistently better than any of reactive algorithms across different prediction interval lengths. Further, we also observe that the profit value for the PLAStiCC algorithm increases as the length of the prediction interval increases. This is because, with shorter prediction intervals, the planned actions for that interval may be already executed before the plan for the future is obtained, and hence leads to a reversal in the actual action and not just the planned action, the former of which incurs additional cost. As the length of the interval increases, fluctuations further in the future are known and hence the planned activities may be reversed before they are enacted and hence lowering the resource cost. In addition, we observe that the heuristics based on current (CAF) and historical average (WAAF) performance







(a

Figure 5. I

perform fairly sir
much lower profi
for the worst cas
cost. Further, Fig
heuristics (lower
PLAStiCC algor
heuristics and he
over the optimiz
algorithm pro-act
advance to accou
new instance at
versions the dela
mismatch till the

This simulatic
PLAStiCC algor
rithms in the pre
Further, it shows
on the length of
as the interval le

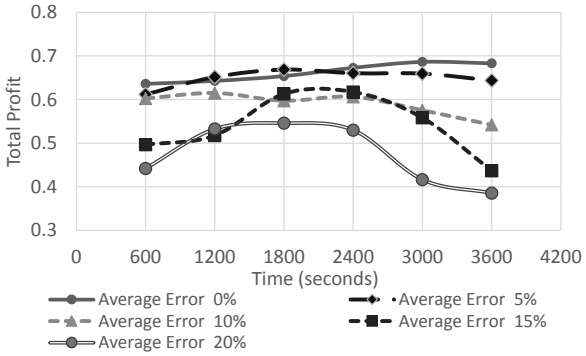


Figure 6. Overall profit of PLAStiCC for different prediction error %, with prediction interval horizon (in secs) on X axis

Predictions with Errors and Biases: Next, we analyze PLAStiCC under a more realistic prediction model conditions, with prediction errors. We compare PLAStiCC against WPO, WPA, and WPP to evaluate their relative strengths.

Figure 6 shows the performance of PLAStiCC for models with prediction errors but no biasing. As before, we observe that the profit of the algorithm for a perfect oracle model (0% error) increases from 0.6 to 0.7 with increase in the prediction interval from $10 - 60mins$. However, we see degrading performance as the error rate increases to 20%. We make two observations. First, for a given prediction interval length the performance decreases, as expected, as prediction error increases. Second, with high error values ($> 10\%$), the performance of the algorithm, for a given error rate, initially increases with the prediction interval length, and then falls with further increase in the interval length (inverse bell curve). The reason for such behaviour is that as we predict farther into the future, the error increases. But as we move into the future, more accurate results are obtained and hence they lead us to reverse actions. While most of these are planned actions, some enacted ones may be reversed too. As the error increases, only prediction very close to the current time will prove accurate and hence the PLAStiCC degenerates to the reactive version.

Finally, we compare PLAStiCC against the reactive versions (WPA, WPO & WPP) which aggregate over values predicted by the model (rather than observed), and are potentially affected by the prediction error. CAF and WAAF are unaffected by prediction errors and hence skipped. Figure 7 shows the overall profit for WPA, WPO, WPP and PLAStiCC for different error rates (X axis) and prediction biases (a-c). A prediction interval of length 20 mins is chosen for these experiments. As seen in the figures, PLAStiCC outperform the others for smaller errors, irrespective of the prediction bias, while its performance degrades as the error increases, since it tries to overfit to the predicted values. WPA, on the other hand performs consistently with different error rates and for different biases. While the WPO algorithm, which assumes an optimistic infrastructure behaviour, performs poorly when the prediction errors are unbiased. However, it performs better than WPA if the predicted values are lower than the actual values. The reason is apparent. With lower predicted values, the best performance values over the interval tend to be closer to the real performance value than the average. On the other hand, WPP performs poorly under all scenarios. This is because the WPP assumes pessimistic infrastructure performance. A single dip in the predicted performance causes it to over-provision resources.

(a) Ove *biased*

(b) Ove *biased*

(c) Overall profit for different algorithms for *high biased* predictor (+5%)

Figure 7. Performance of scheduling with realistic prediction models, having different prediction error % on X axis. Plots differ in prediction bias.

VII. Related Work

A. Continuous Dataflows

Continuous dataflows have emerged as an extension to the Data Stream Managements systems, which were focused on continuous queries over the data streams. These have evolved into a general purpose abstraction that provides strong composition tools for building large scale stream processing applications. Several such stream processing systems have been recently proposed such as S4 [4], IBM InfoSphere streams [17], D-Streams [5], Storm [24], and Time Stream [6]. While these systems are built for large scale stream processing applications and provide distributed deployment and runtime, most of these (with the exception of Time Stream) do not provide automatic, dynamic adaptations to changing data rates or performance variability of the underlying infrastructure, over the application's lifetime.

B. Dynamic Adaptations and Scaling

The notion of "alternates" is similar to flexible workflows [25]. It also draws inspiration from dynamic tasks defined in heterogeneous computing [19] where a task is composed of one or more alternatives with different characteristics, such as resource requirements and dependencies. However, unlike these systems, where the alternate to be executed is selected in advance, in our dynamic continuous dataflow abstraction such decisions are made at periodic intervals based on the changing execution characteristics. Time Stream [6] supports similar dynamic reconfiguration, called *resilient substitution*, which allows replacing a subgraph with another in response to the changing load. This requires dependency tracking and introduces inefficiencies. In contrast, we restrict dynamic adaptation to single processing elements making the process of dynamic reconfiguration independent for each PE and hence allow more flexibility in dynamic adaptation, while restricting the application composition model. Our earlier work has discussed consistency models for enacting such runtime updates [26].

C. Predictive Scheduling

The predictive scheduling approach proposed in the paper follow the generic "scheduler, monitor, comparator, resolver" approach proposed by Nof et. al. [27]. Several such predictive scheduling techniques have been studied on computation grids [28], [12], however in the context of batch task scheduling instead of continuous dataflows. Further, several performance prediction models for the Grid have been proposed to profile workloads as well as infrastructure variations [29]. Although, we do not provide models for performance prediction in the Clouds, we envision that similar performance models may be applicable. PRESS [14] is one such system which uses "signature driven" prediction for variations, with periodic patterns and a discrete-time Markov chain based model for short-term predictions for VMs. They achieve high prediction accuracy with less than 5% over-estimation error and close to zero under-prediction error. We presume existence of such monitoring and prediction algorithm to be used as the basis for our look-ahead scheduling algorithm.

VIII. Conclusion

In this paper, we analyzed the temporal and spatial performance variations observed in public and private IaaS Clouds. We proposed PLAStiCC, a predictive look-ahead scheduling heuristic for dynamic adaptation of continuous dataflows, that responds to fluctuations in stream data rates as well as variations in Cloud VM performance. Through simulations, we showed that the proposed PLAStiCC heuristic results in a higher overall throughput, up to 20%, as compared to the reactive version of the scheduling heuristic, we proposed earlier and improved here, which performs adaptation actions after observing the effects of these variations on the application. We also studied the effect of realistic prediction models with different error bounds and biases on the proposed heuristic, and identified scenarios where the look-ahead algorithm falls short of the reactive one.

References

[1] W. Yin, Y. Simmhan, and V. Prasanna, "Scalable regression tree learning on hadoop using openplanet," in *MAPREDUCE*, 2012.

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010.

[3] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an elastic stream computing platform for the cloud," in *CLOUD*, 2011.

[4] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDMW*, 2010.

[5] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *HotCloud*, 2012.

[6] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *ECCS*, 2013.

[7] D. Zinn, Q. Hart, T. M. McPhillips, B. Ludscher, Y. Simmhan, M. Giakkoupis, and V. K. Prasanna, "Towards reliable, performant workflows for streaming-applications on cloud platforms," in *CCGRID*, 2011.

[8] A. Kumbhare, Y. Simmhan, and V. Prasanna, "Exploiting application dynamism and cloud elasticity for continusous dataflows," in *SC*, 2013.

[9] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *CCGrid*, 2011.

[10] H.-Y. Chu and Y. Simmhan, "Resource allocation strategies on hybrid cloud for resilient jobs," USC, Tech. Rep., 2013.

[11] B. Lu, A. Apon, L. Dowdy, F. Robinson, D. Hoffman, and D. Brewer, "A case study on grid performance modeling," in *ICPDCS*, 2006.

[12] R. M. Badia, F. Escale, E. Gabriel, J. Gimenez, R. Keller, J. Labarta, and M. S. Müller, "Performance prediction in a grid environment," in *Grid Computing, LNCS*, 2004.

[13] D. Spooner, S. Jarvis, J. Cao, S. Saini, and G. Nudd, "Local grid scheduling techniques using performance prediction," *Computers and Digital Techniques*, vol. 150, no. 2, 2003.

[14] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *CNSM*, 2010.

[15] R. P. Weicker, "An overview of common benchmarks," *IEEE Computer*, vol. 23, no. 12, 1990.

[16] R. Wolski, N. T. Spring, and J. Hayes, "The network weather service: a distributed resource performance forecasting service for metacomputing," *FGCS*, vol. 15, no. 5, 1999.

[17] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "Ibm infosphere streams for scalable, real-time, intelligent transportation services," in *SIGMOD*, 2010.

[18] Y. Simmhan, S. Aman, A. Kumbhare, R. Liu, S. Stevens, Q. Zhou, and V. Prasanna, "Cloud-based software platform for data-driven smart grid management," *CiSE*, 2013.

[19] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *JPDC*, vol. 59, no. 2, 1999.

[20] Y. Simmhan, A. Kumbhare, and C. Wickramachari, "Floe: A dynamic, continuous dataflow framework for elastic clouds," USC, Tech. Rep., 2013.

[21] K. Chakraborty, K. Mehrotra, C. K. Mohan, and S. Ranka, "Forecasting the behavior of multivariate time series using neural networks," *Neural networks*, vol. 5, no. 6, 1992.

[22] P. Krause, D. Boyle, and F. Bäse, "Comparison of different efficiency criteria for hydrological model assessment," *Advances in Geosciences*, vol. 5, no. 5, 2005.

[23] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, 2011.

[24] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann, "Stormy: an elastic and highly available streaming service in the cloud," in *EDBT/ICDT Workshops*, 2012.

[25] A. H. H. Ngu, S. Bowers, N. Haasch, T. M. Mcphillips, and T. Critchlow, "Flexible Scientific Workflow Modeling Using Frames, Templates, and Dynamic Embedding," in *SSDBM*, 2008.

[26] C. Wickramachari and Y. Simmhan, "Continuous dataflow update strategies for mission-critical applications," in *eScience*, 2013.

[27] S. Y. NOF and F. HANK GRANT, "Adaptive/predictive scheduling: review and a general framework," *Production Planning & Control*, vol. 2, no. 4, 1991.

[28] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox, "Pace–a toolset for the performance prediction of parallel and distributed systems," *Int. J. HPCA*, vol. 14, no. 3, 2000.

[29] Y. Wu, K. Hwang, Y. Yuan, and W. Zheng, "Adaptive workload prediction of grid performance in confidence windows," *TPDS*, vol. 21, no. 7, 2010.

Efficient Extraction of High Centrality Vertices in Distributed Graphs

Alok Gautam Kumbhare
Computer Science Department
University of Southern California
Los Angeles, CA USA
email: kumbhare@usc.edu

Marc Frincu, Cauligi S. Raghavendra and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, CA USA
email: {frincu, raghu, prasanna}@usc.edu

***Abstract*—Betweenness centrality (BC) is an important measure for identifying high value or critical vertices in graphs, in variety of domains such as communication networks, road networks, and social graphs. However, calculating betweenness values is prohibitively expensive and, more often, domain experts are interested only in the vertices with the highest centrality values. In this paper, we first propose a partition-centric algorithm (MS-BC) to calculate BC for a large distributed graph that optimizes resource utilization and improves overall performance. Further, we extend the notion of approximate BC by pruning the graph and removing a subset of edges and vertices that contribute the least to the betweenness values of other vertices (MSL-BC), which further improves the runtime performance. We evaluate the proposed algorithms using a mix of real-world and synthetic graphs on an HPC cluster and analyze its strengths and weaknesses. The experimental results show an improvement in performance of upto 12x for large sparse graphs as compared to the state-of-the-art, and at the same time highlights the need for better partitioning methods to enable a balanced workload across partitions for unbalanced graphs such as small-world or power-law graphs.**

I. Introduction

With the development of massive online social networks and graph structured data in various areas such as biology, transportation, and computer networks, analyzing the relationship between the various entities forming the network is increasingly important as it allows complex behavior in these large graphs to be explained. The relative location of each vertex in the network helps identify the main connectors, where the communities are, and who is at their core. Centrality indices measure the importance of a vertex in a network [1] based on their position in the network. Betweenness centrality (BC) is one such useful index that is based on the number of shortest paths that pass through each vertex. Formally, BC for a vertex v is defined as the ratio of the sum of all shortest paths that pass through the node v and the total number of shortest paths in the network [2]:

$$BC(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \qquad (1)$$

A simple approach to compute BC is in two steps: first, the shortest paths between all pairs is computed, and second, the length and number of pair-dependencies (i.e. $\delta_{st}(v)$) are computed and hence the betweenness centrality. This has $\mathcal{O}(n^3)$ time, and $\mathcal{O}(n^2)$ space complexity.

Brandes [2] proposed a faster algorithm with $\mathcal{O}(n+m)$ space complexity and $\mathcal{O}(nm)$ time complexity for unweighted graphs (and $\mathcal{O}nm + n^2log(n)$ for weighted graphs). The main idea behind Brandes' algorithm is to perform n shortest path computations and to aggregate all pairwise dependencies without explicitly iterating through n^2 shortest paths to calculate BC for each vertex. To achieve this Brandes noticed that the dependency value $\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$ of a source vertex s on a vertex v satisfies the following recursive relation:

$$\delta_s(v) = \sum_{w, v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}} \left(1 + \delta_s\left(w\right)\right) \qquad (2)$$

where $pred(s, w)$ represents the set of predecessors of w in the shortest path from s to w. By using Equation 2 the algorithm revisits the nodes starting with the farthest one from s, and accumulates the dependency values.

It can be easily seen that the algorithm exposes parallelism at multiple levels. Mainly the shortest path exploration from different source vertices as well as the computation of the individual shortest paths from a given source can be done in parallel. The first shared memory parallel algorithm that exploits the former for exact BC evaluation was proposed by Bader, et al. [3] and improved in [4] by reducing the synchronization overhead by eliminating the predecessor multisets. While this approach has been proven to be much faster, it is architecture specific and assumes a shared memory architecture. On the other hand we explore BC algorithms for large graphs partitioned on commodity clusters as opposed to tightly coupled high performance computing systems since commodity clusters are more accessible especially due to emergence of cloud computing paradigm.

In this paper we propose a partition centric bulk synchronous parallel (BSP) algorithm (§III) that exhibits a more efficient utilization of distributed resources and at the same time shows a significant reduction in the number of BSP supersteps as compared to the vertex centric model. To achieve this we rely on a partition centric approach we and others have proposed earlier [5], [6] and perform an initial partitioning of the graph which allows us to compute shortest paths locally before communicating with other nodes, thus reducing the communication overhead. Following the conclusions of Kourtellis et. al.[7] we focus on approximate computations and extract the top k vertices with highest BC values in order to

speed-up the algorithm. We also propose a light weight pre-processing step that prunes the graph by removing a subset vertices and edges (§IV) that have minimum contribution to the centrality indices of other vertices thus further improving the algorithm runtime. We demonstrate the scalability and tradeoffs for the proposed algorithm as well as the augmented leaf compression version using both real-world and large synthetic data sets (§V). Finally, We also study the impact of partitioning imbalance on different types of graphs and highlight the need for better partitioning techniques.

II. Related Work

Betweenness centrality is a powerful metric used in complex graphs such as social networks as it allows us to identify the key nodes in a graph. These key nodes allow many nodes to quickly connect with each other and at the same time they are critical points in the graph since by eliminating them we could possibly split the graph. As a result much work has been done towards efficiently identifying them since they were first introduced by Freeman [1].

One important question that needs to be answered is "what does centrality refer to?". Borgatti [8] discusses other possible definitions that look at the types of information flows and the frequent paths they take when searching for centrality nodes. While in this paper we restrict ourselves to geodesic paths we argue that our algorithm can be easily applied to any kind of centrality definition as long as the initial partitioning takes that into account.

Brandes' BC algorithm is highly parallelizable due to the way shortest paths are explored and computed. Bader et al. [3] have proposed a shared memory algorithm for exact betweenness evaluation on a dedicated multi-threaded Cray XMT system [9]. Improved versions were proposed in [10] (memory requirements) and [4] (processing and communication requirements). While these algorithms scale on dedicated systems they are not suited for large distributed environments. This is especially important since with the advent of cloud computing the graph datasets may be distributed across compute nodes with relatively higher network latency.

Distributed vertex[11] and partition centric algorithms [12] have also been proposed which exploit the parallelism in computation of the shortest paths. Houngkaew et al. [13] extends the space efficient partition centric approach proposed by Edmonds et. al. [12] by further exploiting the per compute node level parallelism to accumulate dependencies (with explicit mutual exclusion similar to Bader et. al.[3]). While both these approaches are space efficient, we observe that only the compute nodes on the fringe of the shortest paths computation are active and hence leads to under-utilization of the cluster. Our approach improves upon them by computing multiple shortest paths starting from different compute nodes in parallel, albeit, at the expense of additional space requirements.

Because computing BC is extremely costly due to the all-pair shortest path calculations approximate versions have been proposed. The main motivation is that good approximations can be an acceptable alternative to exact scores. In Bader et al. [14] an efficient approximate algorithm based on an adaptive sampling is presented. More recently Kourtellis et al. [7] proposed identifying the top k high betweenness nodes, arguing that the exact value is irrelevant for the majority of applications and that it is sufficient to identify categories of nodes of similar importance. In this paper we too address the aspect of vertices with high BC values and further improve the performance through graph pruning.

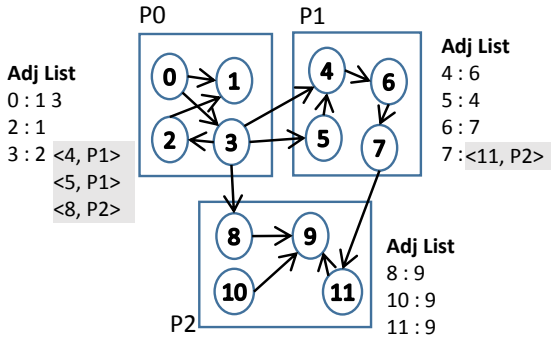
III. Partition-Centric Betweenness Centrality

Vertex centric programming models using BSP[15], [16] for large distributed graph analytics have improved the programming simplicity as well as provided a simple approach for parallelizing graph algorithms. They have been shown to be useful for a large class of graph algorithms, but are prone to performance bottlenecks due to several aspects: low computation to communication ratio because of relatively small amount of work performed per vertex, synchronization required at every superstep and the fact that vertex locality and graph partitioning is ignored by the programming model. Further, the vertex centric models tend to require a large number of supersteps to propagate the information across the graph which further adds to the overall runtime.

To overcome these issues, we and others have proposed partition (or subgraph) centric programming models [17], [5] with the goal of improving the computation to communication ratio by allowing the workers to process an entire partition at a time without any explicit message passing and thus reducing the overhead. This also tends to reduce the number of supersteps required since the messages exchanged between the partitions are available to all the vertices in that partition. The number of supersteps to propagate information is reduced from $O(\text{graph diameter})$ to $O(\text{num. of partitions})$ and is reflected in several graph algorithms such as single source shortest paths [5] which forms the basis of the BC algorithm.

Figure 1a shows a sample graph partitioned into three nodes showing its adjacency list as well as remote vertices that act as pointers to partitions owning those vertices, while Fig. 1b shows a generic approach and programming model for partition centric graph algorithms. The idea is to perform local-computations on the given graph partition, send messages to remote vertices if required and incrementally continue the local work when new messages are received from other partitions. The Do-Local-Compute and Process-Messages functions (fig. 1b) form a core of any partition centric algorithm.

With the development of such new programming paradigms, it is important to explore their strengths and weaknesses with respect to basic kernels in the domain. BC is one such algorithm in graph analytics and we propose a partition centric BC algorithm which improves CPU utilization by increasing parallelism and reduces the number of required supersteps. While Edmonds et. al. [12] proposed a partition centric algorithm (referred to as ΔS-BC), they focus on efficient space utilization and use a Δ-stepping version of the the Single Source Shortest Paths (SSSP). The Δ-stepping shortest paths improves utilization by doing a look-ahead and estimating the distance to the vertices within the look-ahead window using a label correcting approach and updating the distance whenever the estimated value is found to be incorrect. While this approach improves the performance of a single run of the shortest path algorithm, it seems to be limited because the parallization is achieved only by expanding the fringe of the shortest paths algorithm and does not exploit the fact that



(a) Sample graph partition with remote vertices.

```
1: procedure PARTITION-COMPUTE(Partition Graph G, messages<
   targetvertex, list < values >>)
2:     if superstep = 0 then
3:         DO-LOCAL-COMPUTE(G)
4:     else
5:         PROCESS-MESSAGES(messages)
6:         CONTINUE-LOCAL-COMPUTE(G)
7:     end if
8:     for r: remote-vertices do
9:         if condition then
10:            SEND(owner(r), r, msg)
11:        end if
12:    end for
13:    if condition then
14:        VOTE-TO-HALT
15:    end if
16: end procedure
```

(b) Programming Abstraction.

Fig. 1: Partition-centric programming model

multiple shortest paths need to be calculated for computing the centrality values.

We sacrifice in-memory space efficiency in favor of better work distribution among the available processors by running multiple shortest paths starting in different partitions which further increases the amount of local compute and improves the compute to communication ratio. Algorithm 1 (referred to as MS-BC) shows an overview of our approach. The Algorithm is divided into four stages, viz. compute SSSP, find successors, calculate path-counts, and finally update centrality, similar to the vertex centric model BC algorithm proposed by Jonathan et. al. [1]. Each stage in the algorithm in itself follows the partition-compute model (Fig. 1b).

Algorithm 1 Partitioned Betweenness Centrality.

```
1: v ← NEXT-LOCAL-VERTEX
2: COMPUTESSSP(v)    ▷ Each partition start a computation from different source vertex.
3: for Each SSSP Source s in batch do
4:     succ[s] ← COMPUTE-ALL-SUCCESORS(s)
5:     path-counts[s] ← COMPUTE-PATH-COUNTS(s)
6:     UPDATECENTRALITY(C_B, succ[s], path-counts[s]);
7: end for
```

In the SSSP stage each worker independently starts computing the shortest paths from a selected local source vertex, thus utilizing all the nodes in parallel. This runs a modified Dijkstra's algorithm as shown in Alg. 2 which computes tentative distances to the local vertices from the given source and sends a message to the corresponding partition whenever a vertex is identified as a remote vertex. Note that since a partition has no information about a remote vertex and the incoming edges, the distance calculated for the local vertices may be incorrect if there exists a shorter path to the vertex that passes through a remote partition. This entails that the *label assignment* property of the Dijkstra's algorithm no longer holds and hence requires decoupling the path count calculations from the SSSP stage similar to ΔS-BC algorithm due to its *label correcting* approach. Further, as each partition processes incoming messages (Alg. 2) belonging to different paths at the same time, it requires additional aggregate space $O(p \times (n + m))$, where p is the number of partitions and consequently the number of parallel executions of the SSSP algorithm.

[1]https://github.com/Sotera/high-betweenness-set-extraction

Algorithm 2 Partitioned SSSP Computation.

d ← distance map; ▷ default value for each vertex is ∞.
P ← predecessor map;
Q is a priority queue with vertex distance as the key

```
1: procedure LOCAL SSSP(SSSP Source src, Queue Q) ▷ Compute local shortest paths from the given source
2:     while Q not empty do
3:         u ← pop Q;
4:         if owner(u) = current process then
5:             for Neighbor v of u do
6:                 if d[v] > d[u] + 1 then
7:                     d[v] = d[u] + 1;
8:                     DECREASE-KEY(Q, v);
9:                     CLEAR-LIST(P[v])
10:                end if
11:                if d[v] = d[u] + 1 then
12:                    P[v] ← append u;
13:                end if
14:            end for
15:        else
16:            send(owner(u), src, < u, d[u], σ[u], P[u] >);
17:        end if
18:    end while
19: end procedure
```

```
1: procedure PROC. SSSP MSG(SSSP Source src, message list ms)
2:     Q ← empty queue;
3:     min_d[u] ← min_{m∈ms}{m.d where m.u = u};
4:     min[u] ← ⋃ arg min_{m∈ms}{m.d where m.u = u}    ▷ Get all messages with minimum distance for each target vertex u
5:     upd ← 0
6:     for u ∈ min.keys do
7:         if d[u] > min_d[u] then
8:             d[u] ← min_d[u];
9:             DECREASE-KEY(Q, u)
10:            CLEAR-LIST(P[v])
11:        end if
12:        if d[u] = min_d[u] then    ▷ on shortest path?
13:            for Message m ∈ min[u] do
14:                P[u] ← P[u] ∪ m.P
15:            end for
16:        end if
17:    end for
18:    LOCAL SSSP(src, Q)
19: end procedure
```

The compute-all-successors stage traverses the predecessor tree for the shortest paths from each SSSP source in the

Algorithm 3 Recursive Leaf Compression (undirected graph)

```
1: while No Updates do
2:     for v ∈ V do
3:         if EDGECOUNT(v) = 0 then
4:             REMOVEVERTEX(v)
5:         else if EDGECOUNT(v) = 1 then
6:             REMOVEEDGE(Edge)
7:             REMOVEVERTEX(v)
8:         end if
9:     end for
10: end while
```

order of non-increasing distance values and finds successors for the vertices on the shortest paths. The details are omitted for brevity. Once the successors are identified, the paths-count stage performs a similar sweep starting at the source of each shortest path. Finally, the update-centrality stage accumulates the dependency and centrality values using Eqs. 2 and 1 respectively.

IV. Extracting High Centrality Nodes

Given that the exact BC is highly compute intensive and that most applications (e.g., network routing, traffic monitoring etc.) are interested in finding the vertices with highest centrality, algorithms that compute approximate betweenness values [18] and that extract the high centrality vertices [19] have been proposed. The general idea behind these algorithms is to incrementally update the centrality values by computing the shortest paths from a small subset of vertices, called *pivots* until a satisfactory approximation or a stopping criterion is met. This is done in *batches* and the criterion is evaluated at the end of each batch. The batch size has been shown to have direct correlation with the result quality and the algorithm runtime. This approach achieves tremendous speed-ups with high mean precision (MP) while extracting a small set of high centrality nodes [19].

The partition centric BC algorithm can be easily extended to incorporate this strategy by utilizing the master compute function which checks for the terminating criterion at the start of each superstep.

We further extend the notion of approximation and posit that further performance improvements can be achieved with minimal loss of accuracy by performing a pre-processing step which prunes the graph by removing vertices that satisfy the following conditions: (1) the vertices being removed do not exhibit high BC, and (2) removing such vertices has minimum (or equal) effect on the centrality values of other vertices. We observe that the graph's leaves (i.e., vertices with at most one edge) satisfy both these conditions since none of the shortest paths pass through a given leaf vertex (i.e., BC = 0) and it contributes at most one shortest path starting or terminating at that leaf vertex with equal contribution to all the nodes on that path. Hence we use a iterative leaf-compression algorithm (see Alg. 3 for a partition-centric algorithm) as a pre-processing step before running the partition centric approximate betweenness algorithm. Although the formal proof for the error bounds is out of scope of the paper, we present empirical evidence for both performance improvements and relative error in precision in the following section.

V. Evaluations

a) Experimental Setup: To assess the quality and performance of the proposed algorithm (with and without leaf-compression)[2] we compare it against the high centrality node extraction algorithm by chong et al. [19] implemented using the boost MPI version of ΔS-BC[3].

The experiments were run on a HPC cluster consisting of nodes connected with high speed interconnect each with two Quad-Core AMD Opteron 2376 processors and 16GB memory. The experiments were run on a subset of these nodes ranging from 4 to 64 for different graphs with two workers per node (i.e. 8 to 128 workers) each with upto 8GB memory allocated to it. The graphs were partitioned and distributed among the processors using a shared network file system. For the experimental results we do not include the data partition, distribution or loading times since these are consistent across the different studied algorithms and focus on end-to-end algorithm execution time as a performance metric. We use Mean Precision (MP) as the quality metrics as defined by chong et al. [19].

As the quality as well as the performance (due to different convergence rate) of a given algorithm depends on the selection and ordering of pivots, we run each experiment three times with random pivot ordering and report the median MP values and mean runtime values across the three runs.

b) Data Sets: We evaluate the proposed algorithms over a mix of different graph types, both real-world as well as synthetic. The real world data sets includes the Enron email data set, the Gowalla social network graph, and the Pennsylvania road network data set (Tab. I). Random weights between 1 and 10 are generated for each of the edges. We use the real world graphs to analyze the quality of the algorithms, especially with leaf-compression as well as their relative performance. Further, we use a several synthetic sparse graphs to study the performance characteristics and scalability of the proposed algorithms both with respect to increasing graph size and processing cores.

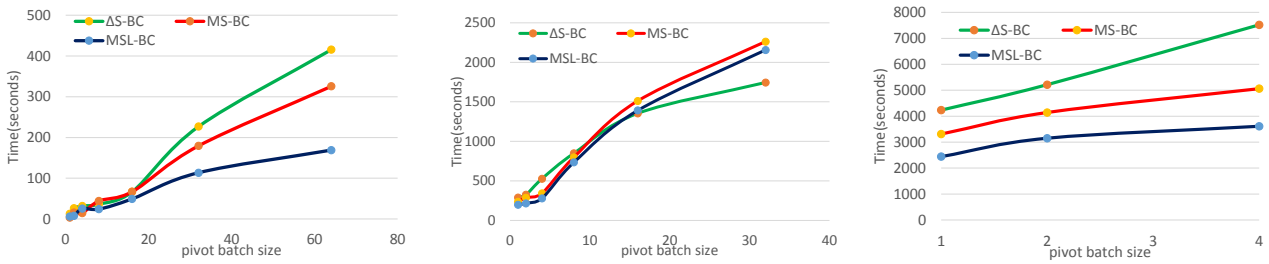
Data Set	vertices	edges	type
Enron Email	36,692	108298	Sparse, Preferential Attachment
Gowalla S/N	196,591	1,900,654	Small World
PA Road N/W	1,088,092	3,083,364	Sparse
Synthetic	100,000	199,770	Erdos-Renyi graph
Synthetic	200,000	801,268	Erdos-Renyi graph
Synthetic	500,000	2,499,973	Erdos-Renyi graph

TABLE I: Data Sets

c) Results: We first compare the quality of the proposed algorithms using real-world graphs with known ground truth. Table II shows the mean precision values for different real-world data set with the pivot batch size varying from 1 to 64 to extract top 32 high centrality nodes. While MS-BC algorithm should give exactly the same results as ΔS-BC ($\Delta = maxedgeweight = 10$) [20] given the same set of pivots, the observed minor variations are due to different pivot selections. Further, we observe that the effect of leaf-compression pre-processing step (MSL-BC) on the algorithm

[2] https://github.com/usc-cloud/goffish/tree/master/high-betweenness-centrality-mpi

[3] http://www.boost.org/doc/libs/1_55_0/libs/graph_parallel/doc/html/betweenness_centrality.html



(a) Enron (b) Gowalla (c) PA Road Network

Fig. 2: Execution times for real world Graphs

quality is minimal and the average error is less than 10% for the Enron data set for higher batch sizes while it performs as well as the others for smaller batch sizes. We see similar trends for the Gowalla s/n and PA road n/w graphs thus empirically validating our hypothesis.

	Enron			Gowalla			CA Road Network		
	MS-BC	ΔS-BC	MSL-BC	MS-BC	ΔS-BC	MSL-BC	MS-BC	ΔS-BC	MSL-BC
1	59.375	56.25	56.25	31.25	28.125	25	25	28.12	21.8
2	50	28.125	56.25	28.125	31.25	25	37.5	34.37	31.25
4	43.75	46.875	56.25	28.125	31.25	28.125	53.125	50	34.375
8	68.75	59.375	53.125	59.375	62.5	46.875			
16	71.875	75	71.875	90.625	84.375	75			
32	71.875	81.25	68.75	87.5	81.25	71.87			
64	90.625	84.375	75						

TABLE II: Mean Precision values for real-world datasets for different batch size (first column refers to the batch size).

Figure 2 shows the average runtime for different real world graphs over a range of batch sizes. We observe that for the Enron and the road n/w data sets, MS-BC algorithm shows significant improvements over ΔS-BC algoritm (˜ 1.5x) due to the better distribution of work among the nodes. Further, the improvement in performance due to leaf compression (MSL-BC) is more pronounced (˜ 2x - ˜ 3x) with higher improvements observed for larger batch sizes. This is primarily because of the improvement in per iteration (one batch of pivots) time due to leaf compression as the number of edges are reduced by 21.19% and 13.74% for the Enron and road n/w data sets respectively. This leads to corresponding reduction in the amount of local work done by each worker which gets accumulated due to the fact that higher batch sizes leads to more pivots being selected.

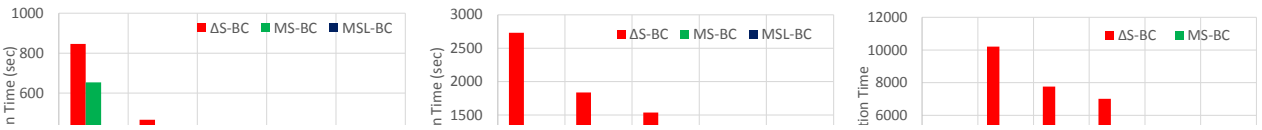
However, we observe that the performance of the algorithm suffers high synchronization penalty for the Gowalla dataset as the batch size (and hence the parallelism) is increased for a fixed set of resources. While the Enron and road network data sets are sparse and balanced (fairly uniform edge distribution), the Gowalla graph exhibits power-law in edge distribution and hence is unbalanced. The goal of the default Metis partitioner [21] is to minimize edge cuts while balancing the number of vertices per node. However, like BC, most graph algorithms' runtime is a function of both the number of vertices and the edges, which is not accounted for by the Metis partitioner and hence leads to imbalance in the runtime of the local components, and in turn causes the synchronization delay during barrier synchronization. For the Gowalla data set we observe that the Metis partitioner produces partitions with 1:30 ratio of edge count between the smallest and the largest partition, where as 1:200 ratio for the local compute (including message processing) function. Further, as we increase the batch size (i.e. parallelism), this effect gets exaggerated as the number of pivots to be processed per partition increases. This implies that the existing partitioning schemes fall short for such graph analytics and techniques such as LALP and dynamic repartitioning[16] have been proposed. However the former is limited in its applicability [16] and the latter incurs run-time performance overhead and hence require further study.

Finally, we study the performance and scalability of the proposed algorithms using a set of synthetic sparse graphs generated using the NetworkX graph generator. Figure 3 shows the execution runtime for sparse graphs of different graph sizes over a range of number of workers. We observe that MS-BC and MSL-BC perform consistently better than ΔS-BC for different graphs sizes and number of workers (as much as 12x performance improvement for large graphs) except for the case of small graphs with large number of workers (e.g. 100K graph with >64 workers). This is because with the increase in number of workers, the partition size decreases and in turn the amount of local work performed. At the same time there is an increase in the number of edge cuts, thus increasing the overall communication. These two factors lead to less than ideal scaling for MS-BC and MSL-BC and hence there is a drop in relative speed-up as shown in figure 4. Further, we observe that MS-BC and MSL-BC fail to completion for large graphs with small number of partitions due to increased memory pressure (e.g. 500k with 4 workers).

This shows both MS-BC and MSL-BC scale well with both the graph size and number of workers. However there exists a break-even point for the number of workers given a graph size beyond which the performance starts to degrade. We also performed similar experiments with other graph types, such as small world and preferential attachment, however, similar to the Gowalla s/n dataset, these graphs are susceptible to partitioning imbalance and hence do not show significant improvement for the proposed algorithms.

VI. Conclusion

In this paper we proposed a partition centric algorithm for efficient extraction of high centrality vertices in distributed graphs that improves the execution time by increasing overall utilization of the cluster and the work distribution. We also prosed a graph compression technique based on leaf-compression that further improves the performance and ex-



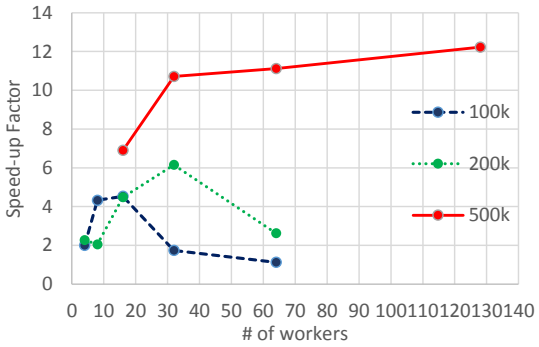


Fig. 4: MSL-BC's speedup relative to ΔS-BC.

perimental results show an improvement of upto 12x for large sparse graphs.

As future work, we will analyze the effect of different partitioning schemes on the algorithm performance and propose a custom partitioner that accounts for graph structure as well as algorithm complexity while partitioning the data.

ACKNOWLEDGMENT

This work was supported by a research grant from the DARPA XDATA grant no. FA8750-12-2-0319. Authors would like to thank Dr. Jonathan Larson of Sotera Defence for the data sets, ground truth results and discussions as well as Charith Wickramaararachi and Yogesh Simmhan for their critical reviews.

REFERENCES

[1] L. C. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, vol. 40, no. 1, 1977.

[2] Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, 2001.

[3] D. Bader and K. Madduri, "Parallel algorithms for evaluating centrality indices in real-world networks," in *Proc. The 35th International Conference on Parallel Processing (ICPP)*, 2006.

[4] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8.

[5] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "GoFFish: A Sub-Graph Centric Framework for Large-Scale Graph Analytics," in *International Conference on Parallel Processing (EuroPar)*, 2014, p. To Appear.

[6] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *PVLDB*, vol. 7, no. 3, pp. 193–204, 2013.

[7] N. Kourtellis, T. Alahakoon, R. Simha, A. Iamnitchi, and R. Tripathi, "Identifying high betweenness centrality nodes in large social networks," *Social Network Analysis and Mining*, vol. 3, no. 4, 2013.

[8] S. P. Borgatti, "Centrality and network flow," *Social Networks*, vol. 27, no. 1, pp. 55 – 71, 2005.

[9] K. P., "Introducing the cray xmt," in *Proceedings Cray User Group meeting*, ser. CUG'07, 2007.

[10] O. Green and D. A. Bader, "Faster betweenness centrality based on data structure experimentation," *Procedia Computer Science*, vol. 18, pp. 399 – 408, 2013.

[11] M. Redekopp, Y. Simmhan, and V. K. Prasanna, "Optimizations and analysis of bsp graph processing models on public clouds," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[12] N. Edmonds, T. Hoefler, and A. Lumsdaine, "A space-efficient parallel algorithm for computing betweenness centrality in distributed memory," in *High Performance Computing (HiPC), 2010 International Conference on*, 2010, pp. 1–10.

[13] C. Houngkaew and T. Suzumura, "X10-based distributed and parallel betweenness centrality and its application to social analytics," in *High Performance Computing (HiPC), 2013 20th International Conference on*, 2013, pp. 109–118.

[14] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Proceedings of the 5th International Conference on Algorithms and Models for the Web-graph*, ser. WAW'07. Springer-Verlag, 2007, pp. 124–137.

[15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010.

[16] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013.

[17] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *PVLDB*, vol. 7, no. 3, pp. 193–204, 2013.

[18] U. Brandes and C. Pich, "Centrality estimation in large networks," *International Journal of Bifurcation and Chaos*, vol. 17, no. 07, 2007.

[19] W. H. Chong, W. S. B. Toh, and L. N. Teow, "Efficient extraction of high-betweenness vertices," in *Advances in Social Networks Analysis and Mining, 2010 International Conference on*. IEEE, 2010.

[20] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of dijkstra's shortest path algorithm," in *Mathematical Foundations of Computer Science 1998*. Springer, 1998.

[21] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, 1998.

Fault-Tolerant and Elastic Streaming MapReduce with Decentralized Coordination

Alok Kumbhare[1], Marc Frincu[1], Yogesh Simmhan[2] and Viktor K. Prasanna[1]
[1]University of Southern California, Los Angeles, California 90089
[2]Indian Institute of Science, Bangalore 560012
Email: kumbhare@usc.edu, frincu@usc.edu, simmhan@serc.iisc.in, prasanna@usc.edu

***Abstract*—The MapReduce programming model, due to its simplicity and scalability, has become an essential tool for processing large data volumes in distributed environments. Recent Stream Processing Systems (SPS) extend this model to provide low-latency analysis of high-velocity continuous data streams. However, integrating MapReduce with streaming poses challenges: first, the runtime variations in data characteristics such as data-rates and key-distribution cause resource overload, that in-turn leads to fluctuations in the Quality of the Service (QoS); and second, the stateful reducers, whose state depends on the complete tuple history, necessitates efficient fault-recovery mechanisms to maintain the desired QoS in the presence of resource failures. We propose an *integrated streaming MapReduce architecture* leveraging the concept of *consistent hashing* to support runtime elasticity along with locality-aware *data and state replication* to provide efficient load-balancing with low-overhead fault-tolerance and parallel fault-recovery from multiple simultaneous failures. Our evaluation on a private cloud shows up to $2.8\times$ improvement in peak throughput compared to Apache Storm SPS, and a low recovery latency of $700 - 1500$ ms from multiple failures.**

***Keywords*-Distributed stream processing; Streaming mapreduce; Runtime elasticity; Fault-tolerance; Big data**

I. Introduction

The MapReduce (MR) programming model [1] and its execution frameworks have been central to building "Big Data" [2], [3] applications that analyze huge volumes of data. Recently, Big Data applications for processing *high-velocity data* and providing rapid results – on the order of seconds or milliseconds – are gaining importance. These applications, such as fraud detection using real-time financial activity [4], [5], trend analysis and social network modeling [6], and online event processing to detect abnormalities in complex systems, operate on a large and diversified pool of streaming data sources. As a result, distributed Stream Processing Systems (SPS) [7] have been developed to scale with high-velocity data streams by exploiting data parallelism and distributed processing. They allow applications to be composed of continuous operators, called *Processing Elements (PEs)*, that perform specific operations on each incoming tuple from the input streams and produce tuples on the output streams to be consumed by subsequent PEs in the application.

The stateful *Streaming MapReduce (SMR)* [8] combines the simplicity and familiarity of the MR pattern with the non-blocking, continuous model of SPSs. SMR consists of mappers and reducers connected using dataflow edges with a key-based routing such that tuples with a specific *key* are always routed to and processed by the same reducer instance. However, unlike the *batch* MR, which guarantees that the reducers start processing only after all mapper instances finish execution and the intermediate data transfer is completed, in SMR a reducer instance continuously receives tuples from the mappers, with different keys interleaved with each other. Hence the reducers need to store and access some *state* associated with an individual key whenever a tuple with that key needs to be processed.

When SMR with stateful reducers are used to process high-velocity, variable-rate data streams with low-latency, a couple of challenges arise:

(1) Unlike batch MR, the system load is not known in advance and data stream characteristics can vary over time – in terms of data rate as well as the reducer's key distribution across tuples. This can cause computational imbalances across the cluster, with overloaded machines that induce high stream processing latency. *Intelligent load-balancing and elastic runtime scaling techniques are required to account for such variations and maintain desired QoS.*

(2) The distributed execution of such applications on large commodity clusters (or clouds) is prone to random failures [9]. Techniques used for fault-recovery in batch MR, such as persistent storage and replication using HDFS with re-execution of failed tasks [10], incur high latency cost of seconds to minutes. Other solutions [11], [12], [13] that rely on process replication are unsuitable for SMR due to their high resource cost and need for tight synchronization. To meet the QoS of streaming applications, *SMR must support fault-tolerance mechanisms with minimal runtime overhead during normal operations while also providing low-latency, parallel recovery from one or more concurrent resource failures*.

In this paper we address these two challenges through an integrated architecture for SMR on commodity clusters and clouds that provides: (1) adaptive load-balancing to redistribute keys among the reducers at runtime, (2) runtime elasticity for reducers to maintain QoS and reduce costs in the presence of load variations, and (3) tolerance to random fail-stop failures at the node or the network level. While existing systems for batch and continuous stream processing partially offer these solutions (see § II), our unique contribution is to build an integrated system for streaming MapReduce that supports above features, by extending well-established concepts such as *Consistent Hashing* [14], Distributed Hash Tables (DHT) [15], and incremental checkpointing [16] to the streaming context while maintaining the latency and throughput requirements in

such scenarios. Specifically, the contributions of this paper are:
(1) We propose a dynamic key-to-reducer (re)mapping scheme based on consistent hashing [14]. This minimizes key reshuffling and state migration during auto-scaling and failure-recovery without needing an explicit routing table (§V-A).
(2) We propose a decentralized monitoring and coordination mechanism that builds a peer-ring among the reducer nodes. Further, a locality-aware tuple peer-backup (§V-B) and incremental peer-checkpointing (§V-C) enables low-overhead load-balancing, auto-scaling as well as fault-tolerance and recovery. Specifically, it can tolerate and efficiently recover from $r \leq x \leq \frac{nr}{(r+1)}$ faults, where r is the replication factor and n is number of nodes in the peer-ring. (§VII).
(3) Finally, we implement these features into the *Floe* SPS and evaluate its low-overhead operations on a private cloud, comparing its throughput with Apache Storm SPS which uses upstream backup and external state for fault-recovery (§VIII). We also analyze the latency characteristics during load-balancing and during fault-recovery, exhibiting a constant recovery time from one or more concurrent faults.

II. Related Work

MR and Batch Processing Systems. MR introduced a simplified programming abstraction for distributed large scale processing of high volume data. A number of MR variations that provide high level programming and querying abstractions, such as PIG, HIVE, Dryad [2], Apache Spark [3], and Orleans [17], as well as extensions such as iterative [18] and incremental MR [19] have been proposed for scalable high volume data processing. However, these fail to consider *runtime elasticity* of mappers and reducers as the workload, and required resources, can be estimated and acquired at deployment time. Further, simple fault-tolerance and recovery techniques such as replicated persistent storage and re-execution of failed tasks suffice since the overall *batch* runtime outweighs the recovery cost. In contrast, SMR requires *runtime load-balancing and elasticity*, as the stream data behavior varies over time, and *low latency fault-tolerance and recovery*, to maintain the desired QoS. These are the focus of this paper.

Scalable SPS and SMR. SPSs (Apache Storm [20], Apache S4 [7], Granules [21], and TimeStream [22]) enable loosely coupled applications, often modeled as task graphs, to process high-velocity data streams on distributed clusters with support for SMR operators. Here, mappers continuously emit data tuples routed to a fixed number of reducers based on a given key. Their main drawback is the limited or lack of support for *runtime elasticity* in the number of mappers and reducers to account for variations in data rates or resource performance. Storm supports limited load balancing by allowing the user to add new machines at runtime and redistribute the load across the cluster, but requires the application to be temporarily suspended, leading to a transient spike in latency. Other systems [23], including our previous work [24], support elasticity and dynamic load-balancing but assume stateless operators or involve costly processes and state migration, and hence unsuitable for SMR.

Other SMR approaches (Spark Streaming [25] and StreamMapReduce [8]) convert the incoming stream into small batches (windows) of tuples and perform batch MR within each window; the reducer state is embedded in the batch output. Runtime elasticity can be achieved at the window boundaries by varying the number of machines based on the load observed in the previous window. However, this has two downsides. First, depending on the window size, the queuing latency of the tuples can grow to tens of seconds [25]. And second, since these systems use a simple hash-based key-to-reducer mapping function, scaling in/out causes a complete reshuffle of the key mappings, incurring a high overhead due to the large state transfer required.

Fault-tolerance and state management in SPS. Traditional SPSs support fault-tolerance using techniques such as *active replication* [26] which rely on two or more copies of data and processes. Recent systems such as S4 [7] and Granules [21] provides partial fault-tolerance by using a centralized coordination system such as Zookeeper. They offer automatic fail-over by launching new instances of the failed processes on a new or standby machine. They also perform periodic, non-incremental checkpointing and backup for individual processes, including the tuple buffer, by using synchronous (de)serialization which requires pausing the process during checkpointing. Further, they use an external in-memory or persistent storage for backup. These approaches add considerable overhead and lead to high processing latency during checkpointing as well as recovery. In addition, they do not guarantee against tuple loss as any in-flight and uncheckpointed tuples may be lost during the failure.

Systems such as Storm [20], Timestream [22] guarantee "atleast once" tuple semantics using a combination of upstream-backup and an explicit acknowledgment tree, even in the presence of multiple node failures. Trident, an abstraction over Storm improves that to "exactly once" semantics using an external, persistent coordination system (zookeeper). However, neither of these support state recovery, and any local state associated with the failed processes is lost. A user may implement their own state management system using persistent distributed cache systems (DHT, Zookeeper) but this increases the complexity and processing overhead per tuple. SEEP, [27], [28] integrate elastic scale-out and fault-tolerance for general purpose SPSs using a distributed, partitioned state and explicit routing table. Their solution, while applicable to SMR, incurs higher overhead during load-balancing, scaling and fault-recovery as it fails to take advantage of the key grouping and state locality property of the reducers. This causes significant reshuffling of the key-to-reducer mapping.

Martin et. al. [29] proposed a streaming map reduce system with low-overhead fault tolerance similar to our proposed system. The key distinguishing factor is our support for runtime elasticity and load-balancing to handle variability in data streams observed at runtime. Further, their system enables deterministic execution (*exactly-once semantics*) and relies on the virtual synchrony [30] method for synchronization and synchronous checkpointing at the end of each epoch (check-

point interval) which increases overall latency and further requires total ordering of messages from different mappers, which is difficult to achieve.

On the other hand, our approach provides efficient state management and fault-recovery, and guarantees "atleast once" tuple semantics with no tuple loss during failure. It combines asynchronous, incremental peer-checkpointing for reducer state and peer tuple backup with intelligent collocation to reduce the recovery overhead by minimizing state and tuple transfer during recovery. We also employ a decentralized monitoring and recovery mechanism which isolates the fault to a small subset of reducers while the rest can continue processing without interruptions.

III. Background

Streaming MapReduce. SMR [8], [31] extends the batch MR model by leveraging stateful operators in SPS and using a key-based routing strategy for sending tuples from mappers to reducers. The mapper is a stateless operator that transforms the input tuples and produces one or more output tuples of the form $t = \langle k_i, v \rangle$. Unlike batch MR, SMR relaxes the "strict phasing" restriction between map and reduce. Hence, the system does not need to wait for all mappers to complete their execution (i.e., produce all tuples for a given key) before starting the reducers. Instead, tuples are routed to the matching reducer as they are emitted and the reduce function is executed on each incoming tuple, producing continuous results.

Unlike the batch MR, where reducers are stateless as they can access all the tuples for a given key during execution, in SMR, *reducers must be stateful*. As tuples with different keys may arrive interleaved at a reducer, a single reducer will operate on different keys, while maintaining an independent state for each key. Specifically, the reducer function takes a tuple, and a state associated with the given key, performs a stateful transformation and produces a new state with an optional output key-value pair, i.e. $\mathbb{R} : \langle k_i, v \rangle, s_j^{k_i} \rightarrow [\langle \widehat{k_i}, \widehat{v} \rangle,] s_{j+1}^{k_i}$, (for e.g., see Fig. 9 in Appendix).

Execution Model. The SMR execution model is based on existing well established SPSs such as Storm, S4 [7], and Floe [32]. Figure 1 shows an example of a SMR application on five hosts (physical or virtual). The mapper and reducer instances are distributed across the available hosts to exploit inter-node parallelism. Each host executes multiple instances in parallel threads, exploiting intra-node parallelism. Although mappers and reducers may be collocated (preferred), for simplicity we assume that they are isolated on different hosts. We thus refer to a node hosting a number of mappers or reducers as a *Mapper node* or *Reducer node*, respectively. Each mapper consumes incoming tuples in parallel, processes and emits tuples of the form $\langle k_i, v \rangle$ where k_i is the key used for routing the tuple to a specific reducer. Given the stateless and independent nature of the mappers, simple load-balancing and elastic scaling mechanisms [33], [24] are adequate and are not discussed here. Instead, we focus on *load-balancing, elasticity, and fault-tolerance* only for the reducer nodes.

The tuples are reliably transmitted to the reducers with assured FIFO ordering between a pair of mapper and reducer.

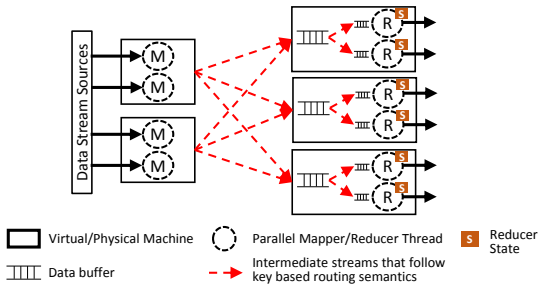


Fig. 1. Distributed SMR execution model.

However, given that the several mappers may process and produce tuples in parallel, the *total ordering* among the tuples generated by them, even for a single key, is undefined. Hence, the reducer logic should be agnostic to the order in which the tuples arrive. This constraint is similar to the traditional MR, as well as other stream processing systems. For applications that require specific ordering, techniques such as used by Flux [26] or Virtual Synchrony may be used but are out of scope.

Fault Model. We only consider and guard against fail-stop failures. These occur due to hardware failures, critical software bugs or planned upgrade and reboot in the cluster or cloud environment. In such cases we assume that the acquired resource becomes unavailable and we lose all the buffered data and state associated with that host. Further, individual network link failures are treated as host failures and the fault-tolerance techniques reused. Recovering the lost data and state by replaying tuples from the beginning is not possible as it would cause an unacceptable latency, and hence requires efficient state-management and low-latency fault-tolerance techniques to recover the lost state to continue processing. Using simple techniques that store the state in an external distributed shared, persistent [7], [20] memory (DHT, HDFS, databases) would also suffer from high processing and recovery latency since these systems lack a notion of locality and the state updates and recovery require expensive read/write operations from the external system.

IV. Goals of Proposed System

Performance and Availability. In SPSs, QoS is defined not only in terms of overall system availability and its capability to eventually process all incoming tuples, but is also measured in terms of *average response time* – the latency between when a tuple is generated by the mappers and when it is processed by the reducer, including the corresponding state update. Brito et al. [8] identified several classes of applications with expected response time ranging from minutes or hours for traditional batch data analysis, to less than 5 ms per incoming tuple for real-time applications. Here we focus on fast streaming applications with average response times ranging from $10 - 500$ ms, but real-time applications with millisecond or sub-millisecond response time is out of scope.

The response time includes *transmission*, *queuing*, and *processing* delays. While the transmission and processing delays are a function of the hardware and application logic, the queuing delay is a function of the variable load on the host. The queuing delay may be reduced by transferring its current

load to other hosts or newly acquired resources. Such *load transfer* should incur low overhead and not interfere with the regular system operations. Application performance should be maintained during failure and fault-recovery procedure while reducing recovery overhead. These low-latency requirements dictate the design of our system and preclude the use of persistent storage for fault-tolerance. Replicated in-memory distributed hash tables (DHTs) can persist state for failure-recovery but are costly during normal operations for accessing and updating the distributed state since they are not sensitive to the locality of the state on reducer(s).

Deterministic Operations. We assume that the reducer is "an order agnostic, deterministic operator", similar to batch MR, i.e., it does not require tuples to arrive at a certain time or in a particular order to produce results, as long as each tuple is processed exactly once by the reducer. This implies that the application can be made deterministic if we ensure the following: (D1) the state associated with a key at each reducer is maintained between successive executions, i.e., the state is not lost, even during failures; (D2) none of the tuples produced by the mappers are lost; and (D3) none of the tuples produced by the mappers are processed more than once for two different state updates. The last condition might occur during failure recovery or when shifting the load to another reducer (§VI). A tuple is non-deterministic if used for multiple different state transformations, a case which may occur if: (1) the state, or part of it, is lost during recovery (which should never occur as it violates D1), or (2) the reducer instance is unable to determine if the replayed tuple was processed earlier and applies the update again on the recovered state.

Achieving strict determinism that satisfies D1, D2, and D3 is expensive and requires complex ordering and synchronization [26]. In this paper, we relax these and allow "atleast-once" tuple semantics, instead of "exactly-once", i.e., we ensure that D1 and D2 are strictly enforced, but some of the tuples may be processed more than once by the reducers, thus violating D3. Although this limits the class of applications supported by our system, this is an acceptable compromise for a large class of big data applications and can be mitigated at the application level using simple techniques such as *last seen* timestamp.

V. Proposed System Architecture

To achieve the dual low-latency goals of response time performance and fault-recovery under variable data streams, we need to support adaptive load-balancing and elastic runtime scaling of reducers to handle the changing load on the system. Further, we need to support low-latency fault-tolerance mechanism that has minimal overhead during regular operations and supports fast, parallel fault-recovery.

We achieve adaptive runtime load-balancing and elastic scaling of stateful reducers by efficiently *re-mapping* a subset of keys assigned to each reducer, at runtime, from overloaded reducer nodes to less loaded ones. However, the reducer semantics dictate that all the tuples corresponding to a particular key should be processed by a single reducer which conflicts with this requirement. Hence, to transparently meet these semantics, we efficiently manage the state associated with the overloaded reducer and migrate the *partial* state corresponding to the re-mapped keys.

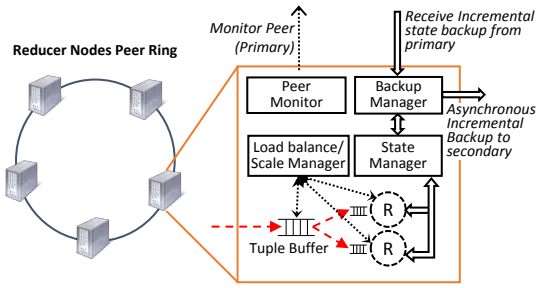


Fig. 2. Reducer node components.

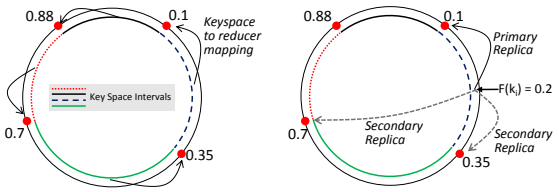
In addition, to support efficient fault-tolerance and recovery, we backup tuples with minimal network and memory overhead, and perform asynchronous incremental checkpointing and backup such that the tuple and state backup corresponding to a specific key are collocated on a single node. The asynchronous checkpointing ensures that performance of regular operations is not affected, and state and tuple collocation ensures that the recovery from failure is fast, without requiring any state or tuple transfer over the network.

The proposed system design integrates these two approaches by building a reducer *peer-ring* as shown in Fig. 2. We overlay a decentralized monitoring framework where each reducer node is identical and acts both as a *primary node* responsible for processing incoming tuples, as well as a *secondary node* responsible for backing up its neighbor's tuples and state as well as to monitor it for failures. A reducer node consists for several components (Fig. 2): a state manager responsible for maintaining the local partitioned state for each key processed by the reducer; a backup manager responsible for receiving and storing state checkpoints from its peers as well as to perform asynchronous incremental checkpointing for the local state; a load-balancer/scaler responsible for monitoring the load on the reducer and perform appropriate load-balance or scaling operations; and a peer monitor responsible for monitoring its peer for failure and perform recovery actions. We discuss these techniques in detail and describe the advantages offered by the peer-ring design in meeting our system goals.

A. Dynamic Re-Mapping of Keys

The shuffle phase of batch MR uses a mapping function $F = \text{HASH}(k) \bmod n$, where k is the key and n is the number of reducers, to determine where the tuple should be routed. The shuffle phase is performed in two stages. First, the mapper applies F to each outgoing tuple and sends the aggregated results to the designated reducer. Once all mappers are done, each reducer aggregates all the tuples received for each unique key from several mappers, and then executes the reducer function for each key over its complete list of values. Scalable SMR is similar except that the tuples are routed immediately to the corresponding reducer which continuously processes the incoming tuples and updates its state (§III).

However, when performing elastic scaling, the number of reducers can change at runtime. So the above mapping



(a) Consistent hashing. (b) Peer-backup for r=2.

Fig. 3. Consistent hashing examples with 4 reducers.

function F becomes infeasible since adding or removing even a single reducer would cause a shuffle and re-map of many of the existing keys to different reducers, leading to a large number of state migrations between the reducers. This $\mathcal{O}(|K|)$ remapping, where $|K|$ is the number of unique keys, will introduce a high processing latency. Existing SPSs, such as Storm, S4, SEEP and Granules, are limited in their runtime scale in/out due to the use of such a hash function.

To overcome this, we rely on *consistent hashing* [14] for key-to-reducer mapping whose *monotonocity* property ensures that adding or removing reducers, only affects a small portion, $\mathcal{O}(\frac{|K|}{n})$, of the key set, needing less state migrations and incurring a smaller latency. The idea is to assign each of the reducers a unique *"token"* $\in [0, 1]$ to form a *ring* (Fig. 3a). The keys are always mapped to $[0, 1]$ using a well-distributed hash function. We then select the closest reducer in the counter-clockwise direction of the ring from that point. The complexity of mapping the key to a reducer is $\mathcal{O}(1)$.

Whenever a reducer is added, it is assigned a new unique token which puts it between two existing consecutive reducers, dividing the keys space mapped to the original reducer into two subsets, one of which is mapped to the newly added reducer without affecting key mapping for any of the other existing reducers. Similarly whenever a reducer is removed (e.g., due to a fault) its keys are dynamically mapped to the next reducer on the ring in the counter-clockwise direction.

The basic consistent hashing algorithm assigns a random position for each of the nodes on the ring possibly leading to non-uniform load distribution. It is also vulnerable to changes in the system's load due to the variations in data rates and key distribution over time. A virtual nodes approach [34] addresses only the initial non-uniform load distribution issue. Instead, we use a dynamic approach, similar to Cassandra, that allows a node to move along the ring at runtime in response to the variations in the system load.

B. Peer-Backup of Tuples

To achieve efficient backup of incoming tuples to a reducer, we again use a variation of *consistent hashing* that assigns each key to $r+1$ contiguous buckets on the ring onto which the tuple is backed up; r is the tuple replication factor. Fig. 3b shows a sample configuration with 4 buckets and $r = 2$. A key k_i is hashed to the interval $[0, 1]$ and the *primary replica* is selected by finding the nearest neighbor on the ring in the counter-clockwise direction as before. In addition, $r = 2$ neighboring buckets are chosen as *secondary replicas* by traversing the ring in the clockwise direction. The mapper then sends the tuple to all 3 of the nodes (1 primary, 2 secondary), using a reliable multi-cast protocol (e.g., PGM) to minimize network traffic.

On receiving a tuple, each node checks if it is the primary by verifying if it appears first in the counter-clockwise direction from the tuple's position on the ring. If so, the tuple is dispatched to the appropriate reducer thread on the machine for processing. Else, the node is a secondary and the tuple is backed-up in-memory, to be used later for load-balancing or fault-recovery. Note that each node can determine if it is a primary in $\mathcal{O}(1)$ time, and needs to navigate to at most r clockwise neighbors. Note that it is important to clear out tuple replicas which have been successfully processed to reduce the memory footprint. This eviction policy is discussed in V-D.

C. Reducer State Model and Incremental Peer-Checkpointing

The state model must support (1) *partial state migration*, i.e., migrating the state associated with only a subset of keys at a reducer, and (2) *reliable, incremental and asynchronous checkpointing* [35], [16], [36], i.e., the state must be deterministic, not concurrently updated by the reducer during checkpointing, and without the need to pause the reducers during this process.

Partial states can be managed by decoupling the state from reducer instance, and partitioning it based on the individual keys being processed by the reducer (Fig. 4). This allows incremental backup only for the keys processed and updated during the last checkpointing period. We can further reduce the size of the incremental backup by restricting the state representation to a set of key-value pairs (*Master State* in Fig. 4). Given this two-level state representation, an incremental backup can be obtained by keeping track of reducer keys updated during the latest checkpoint and the set of key-value pairs updated for each of the reducer keys.

We divide the state associated with each key into two parts, a *master state* and a *state fragment*. The former represents a stable state, i.e., a state that has been checkpointed and backed up onto a neighbor. The latter represents the incrementally updated state which has to be checkpointed at the end of this checkpoint interval. After the fragment is checkpointed, it can be merged into the master state and cleared for the next checkpoint interval. This allows efficient incremental checkpointing, but can lead to *unreliable* checkpoints as the state fragment may be updated by the reducers during the checkpointing process. We can avoid this by pausing the reducers during checkpointing, but it incurs high latency. Instead we propose an *asynchronous* incremental checkpointing process.

Here, we divide the state fragment into two mutually exclusive sub-fragments: *active* and *inactive*. The active fragment is used by the reducers to make state updates, i.e., the key-value pairs are added/updated only in the active fragment, while the inactive fragment is used only during checkpointing, as follows. At the end of a checkpoint interval, the active fragment contains all the state updates that took place during that checkpoint interval, while the inactive fragment is empty. To start the checkpointing process, the state manager *atomically* swaps the pointers to the active and inactive fragments and the

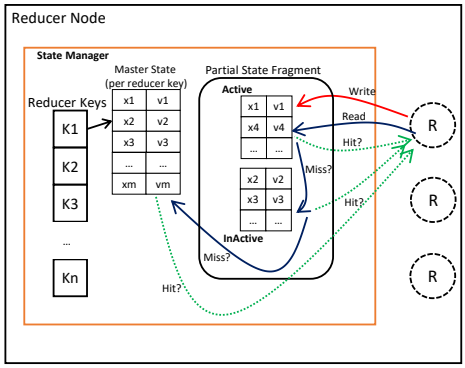


Fig. 4. State representation with master state and partial state fragments.

corresponding timestamp is recorded. The reducers continue processing as usual and update the state in the active fragment, while the inactive fragment contains a *reliable* snapshot of updates that occurred in the previous interval. The inactive fragment is then asynchronously serialized, checkpointed, and transferred to the backup nodes using a multi-cast protocol. After completion, the inactive fragment is merged with the master state and is cleared for the next checkpoint cycle.

With respect to freshness, the active fragment contains the most recent updates, followed by the inactive fragment and finally the master state. Thus, a "state read" request for a key-value pair from the reducer is first sent to the active fragment, then the inactive fragment, and finally the master state until the corresponding key-value pair is found (Fig. 4). While this involves three seeks, it can be mitigated with $\mathcal{O}(1)$ data structures like hash tables.

The combination of partitioned state and active/inactive state fragments allows the reducer thread to continue processing incoming tuples and update the state in the active fragment during the checkpointing process without any conflicts or interruptions thus minimizing the overhead (See § VIII). As with tuple-backup, we follow an optimistic checkpointing process where the primary host does not wait for an acknowledgment from any of the backup hosts, letting reducers execute uninterrupted. This optimistic replication works as long as atleast one backup node is available. We rely on the peer ring to determine the hosts to be used for backup to ensure tuple and state collocation for fast recovery.

D. Tuple Ordering and Eviction Policy

The tuple eviction policy determines which tuples can be safely removed from the backup nodes such that the state associated with the failed host can be recovered by replaying the remaining tuples and updating the checkpointed state. Tuple eviction allows us to keep a low memory footprint.

The proposed eviction policy allows for a small subset of tuples to be processed more than once. We assume that all the mappers are loosely time synchronized and that the maximum time skew between any two mappers is less than the maximum network latency (L_n) to transfer tuples from mappers to reducers. Such assumption have been used in systems such as Flux [26] to identify missing messages and is practically justified since using NTP an error bound within few μ-seconds may be achieved and network latency is typically observed in the milli-seconds range.

Each mapper emits tuples with an associated timestamp. We assume a reliable multi-cast protocol to ensure that emitted tuples are delivered in FIFO order. The primary host then processes the tuples in the order they arrive from different mappers and marks the state update with the latest timestamp of the tuples that affect the current state. Given the time skew between mappers and the maximum network latency, L_n, a tuple with lower timestamp may be received and processed at a later point. In this case the timestamp mark on the state is not updated.

The backup hosts also receive tuples from different mappers for backup. These hosts store the backup tuples in decreasing order of their timestamps. Maintaining this order is not costly since the tuples are usually received in an increasing order of timestamp (except for few due to clock skew and network latency). Whenever a checkpoint is received from the primary, the secondary host retrieves the associated timestamp T_s and evicts all tuples with a timestamp less than $T_s - 2L_n$ (instead of tuples with TS less than T_s) since tuples with an earlier timestamp may arrive later and may have not been processed yet by the primary node. This leaves some potential tuples in the backup which may have already been processed and reflected in the checkpointed state. If the primary host fails, the backup node tries to recover the state by replaying the backup tuples and updating the checkpointed state. This may lead to some tuples being processed twice, violating condition D3. Certain measures can be implemented at the application level to handle this scenario but are out of our scope. As long as the time skew is bounded by network latency, L_n, there will be no tuple loss during failure, hence condition D2 will be met. Note that the issue of variability in network latency over time can be mitigated in one of the two ways: first, by setting a much high value of the threshold (L_n) compared to the observed maximum network latency (10-20x) such that any latency higher than the threshold may be considered as network failure. However, a side effect of high threshold is that it significantly increases the number of messages that may be replayed during a fault recovery or scaling-in scenario. Second, we can monitor the network latency over time and update the threshold ($2 \times L_n$) to reflect the variations in network latency. Note that this can be achieved efficiently in a decentralized manner since each of the nodes in the ring can monitor the latency and use a gossip protocol to propagate the changes to other nodes in the ring. We use the first approach for system evaluation in Section VIII.

VI. Adaptive Load Balancing and Elasticity

Because of fluctuations in the data streams two scenarios can occur. First, the number of keys mapped to a particular interval (i.e., on a reducer node) may fluctuate at runtime, and second, the rate of generated tuples associated with a specific key may vary at runtime. Both scenarios can lead to processing load imbalances among the reducer nodes.

The proposed architecture supports adaptive load balancing by leveraging the fact that the neighboring nodes (secondary

instances) associated with a key already possess a recently cached state (peer-checkpointing) as well as the unprocessed tuples (peer-backup) associated to that key.

We use a simple load-balancing strategy where each host monitors its buffer queue length (q_L) and the overall CPU usage (c) while processing the incoming tuples. A host is said to be overloaded if $q_L \geq \tau_q^{high}$ *AND* $c \geq \tau_c^{high}$. While this strategy is prone to oscillation in resource allocation if frequent variations are observed, we have previously proposed robust runtime adaptation algorithms that not only consider the current system load but also observe data patterns and use predictive techniques to avoid such oscillations as well as to minimize resource cost [33], [24], which can be applied here.

An overloaded node (A) negotiates with its clockwise neighbor (B) (i.e., the first backup host) to see if it can share some of the load (i.e., $q_L \leq \tau_q^{low}$ *AND* $c \leq \tau_c^{low}$ for that node). If so, it requests B to expand the primary interval associated with that node by moving in the counter-clockwise direction on the ring. Figure 10 in Appendix shows a sample configuration and the transformation that happens during the load balancing phase. The distance by which the backup node should move can be determined by the exact load on the two systems and the current distribution of keys along the path. However, for simplicity, we assume that the backup node always moves by half the distance between the two at a time.

Node B then trivially recovers the state associated with the transferred keys that was checkpointed earlier by A. It then replays and processes the tuples buffered in its backup (i.e., those tuples not reflected in the currently checkpointed state) and starts processing the new tuples associated with that interval. The backup node now becomes the primary for the given interval and relieves the load on the original primary (A) by taking over some of its load. It also notifies the mappers about the change using multi-cast. Note that the backup node need not wait for this to be delivered to the mappers, since, being the secondary node, it already receives all the relevant tuples. On receiving the notification, the mappers will update their ring and stop sending the tuples to A and start sending it to B and the load-balancing will be completed.

The adaptive load-balancing technique allows an overloaded node to negotiate with its immediate neighbor to offload some of its work. However, if the neighboring host does not have spare cycles and is working to its capacity, it can in turn request its neighbor to offload before accepting the load from the primary. If none of the hosts can accept additional load, the primary host will elastically scale out as follows.

Scaling Out. Scaling out involves provisioning a new host, putting the new host on the ring such that it offloads some of the work from the overloaded primary node, and transferring the corresponding state on to it. To minimize the overhead we start the state transfer in background only state corresponding to the keys that will be offloaded. We also start sending (duplicating) tuples associated with the keys to the new host for backup (using the multi-cast protocol, hence without the additional hop). During this process, the primary node continues to process the incoming tuples as before (albeit at a slower rate due to overload) and update the active state fragment as before. Once the master state transfer is completed, the previously described checkpointing process is performed on the backup nodes as well as on the newly acquired node. This step ensures that the new host has the latest state and that the processed tuples are evicted from its backup buffer. Finally, the new host is put on the peer-ring mid-way between the overloaded node and its neighbor and its r neighbors are informed about the change (see Fig. 11 in Appendix). The new host thus takes over some work from the overloaded node.

Scaling In. A primary host can be scaled in if that host along with its clockwise neighbor are lightly loaded (i.e., $q_L \leq \tau_q^{low}$ *AND* $c \leq \tau_c^{low}$ for both hosts). The primary host can offload all of its load to the neighbor and can return to the resource pool and be available for other requirements or shutdown to conserve cost and energy. The process is similar to the load-balancing process including the checkpoint, tuple replay, and state recovery, except that the neighbor's interval is now expanded to cover the primary's entire interval by setting its token to be equal to the primary's token and removing the primary from the ring (Fig. 12 in Appendix).

VII. Fault-Tolerance

A system is said to be r-fault-tolerant if it can tolerate r arbitrary faults (host failures cf. §III) and can resume its operations (i.e., satisfy D1, D2) without having to restart/redeploy the entire application. The proposed system can tolerate at most r failures in *consecutive* neighbors on the ring since the state and the tuples are backed up on $r+1$ hosts. However, it can tolerate more than r failures if the failures do not occur in consecutive neighbors on the ring. Specifically, in the best case scenario, it can tolerate up to $x = \frac{nr}{(r+1)}$ node failures as long as the faults do not occur in more than r consecutive neighbors (where r is the replication factor and n is number of nodes). For example, with $r = 1$, the system can still be functional (i.e., no state or tuple loss) even if every alternate node on the ring (i.e., $n/2$ nodes) fail simultaneously.

Recent large scale studies [37] have shown that a significant portion of failures observed in a datacenter are spatially collocated (e.g., overheating, rack failures, cluster switch failures) and that the datacenter can be divided into several "fault zones" such that the probability of simultaneous failures for machines in different fault zones is lower than that for machines within a single fault zone. This is also the basis for the "fault domains" or "availability zones" feature provided by Microsoft Azure and Amazon AWS which provide atleast 99.95% availability guarantees if VMs are placed in distinct fault zones. We can exploit this property and place neighbors on the ring in distinct fault zones (Fig. 13 in Appendix) to achieve higher fault-tolerance.

Fault-tolerance involves two activities: fault-detection and recovery. To achieve the former in a decentralized manner we overlay a monitoring ring on top of the peer-ring where each host monitors its immediate **counter-clockwise** neighbor (i.e., a secondary node which holds tuple and state backup for a subset of the key space monitors the primary node). Whenever

a fault is detected, to initiate fault recovery, it contacts the one-hop neighbor of the failed node to see if that node is still alive and continues this process until r hops or a live host is found. Note that at this point the backup host will have the checkpointed state and tuples for all failed nodes. Hence, it can employ a procedure similar to the *scale in* process to take over the load from one or more ($\leq r$) failed nodes. As before, no state transfer or additional tuple transfer is required.

The fault-recovery process by itself does not provision additional resources but uses the backup nodes for recovery so as to minimize downtime. However, during or after the takeover, if the backup node becomes overloaded, the adaptive load-balance or scale-out process will be initiated to offload some of its work without interrupting the regular operations.

VIII. Evaluation

We implemented a prototype of our proposed architecture on top of Floe SPS [32]. It provides a distributed execution environment and programming abstractions similar to Storm and S4. In addition, it has built-in support for elastic operations and allows the user to add or remove parallel instances of (stateless) PEs at runtime using existing or acquired resources. We extend it to support elasticity as well as fault-tolerance for stateful reducers via the proposed enhancements.

The goal of the experiments is not to study the scalability of the system on hundreds of nodes, but to evaluate the dynamic nature of the proposed system and the overhead it incurs. The setup consists of a private Eucalyptus cloud with up to 20 VMs (4 cores, 4GB RAM) connected using gigabit Ethernet. Each map/reduce node holds at most 4 corresponding instances. We use a streaming version of the *word frequency count application* that keeps track of the overall word frequency for each unique word seen in the incoming stream as well as a count over a different sliding windows (e.g. past 15 mins, 1 hr, 12 hrs etc.) which represents recent trends observed in the stream. Such application may be used in analysis of social data streams to detect top "k" trending topics by ranking them based on their relative counts observed during a recent window. In such applications, the exact count for each of the topics is not required since the relative ranking among the topics is sufficient. As a result, the system's atleast-once semantics for message delivery is acceptable for the application. We emulate the data streams by playing the text extracted from the corpus of text data from the Gutenberg project. Each mapper randomly selects a text file and emits a stream of words which is routed to the reducers. To demonstrate various characteristics, we emulate: (1) variations in overall data rate by dynamically scaling up/down the number of mappers, and (2) variations in data rate for a subset of keys (load imbalance), by using streams that repeatedly emit a small subset of words.

We synchronize the VMs using NTP and get a loose synchronization bound within few μsec. Further, we determine the maximum network latency (L_n) to be around 1ms by executing a number of ping requests between the VMs. Nonetheless we use a conservative estimate of 15ms for our experiments and a value of $2 \times L_n = 30$ms as a bound to evict tuples from the backup buffer to account for variations in the network latency and potential time drift observed over time.

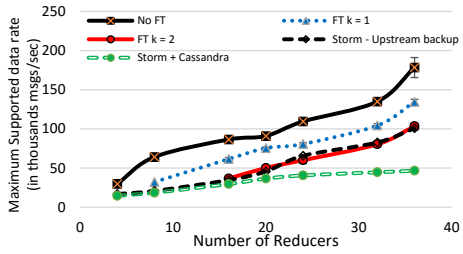
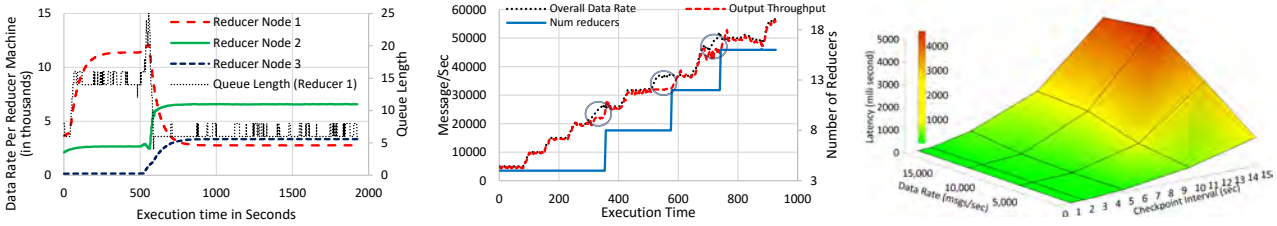


Fig. 5. Achieved Peak throughput for different number of reducers.

A. Empirical Results

We first evaluate the system under static configurations to determine the overhead due to the checkpointing and backup mechanisms. We fix the number of VMs and reducers at deployment time and progressively increase data rates to determine the maximum achievable cumulative throughput (processing rate of the reducers). We examine the system under different tolerance levels $r = 0, 1, 2$, where 0 indicates that no fault-tolerance mechanisms are in place. We compare our system against two variations of the application deployed using Storm SPS. The first uses Storm's upstream-backup feature with explicit acknowledgments ensuring that no tuples are lost during failure. However, the state is stored locally and may be lost if the corresponding node fails. The second relies on an external distributed reliable in-memory storage (Cassandra) to store the state associated with each key. This version provides fault-tolerance and recovery as well as protection against tuple loss similar to the proposed system, but incurs significant overhead. Fig. 5 shows the peak throughput achieved by these systems as a function of number of reducers. Following key observations can be made from Fig. 5: (1) The peak throughput achieved by Floe at $r = 0$ is consistently higher than others due to minimal overhead and it drops by around 15% as we increase the tolerance level. This is expected since higher r values require additional tuple and state transfer and adds to the load on secondary nodes. (2) Floe achieves higher throughput than both versions of Storm giving around 2.8x improvement for $r = 1$ compared to Storm with state backup using Cassandra due to high latency incurred during state access. (3) Finally, we observe that Floe scales (almost) linearly as we increase the number of resources, while Storm's peak throughput flattens out after a certain point due to the bottleneck caused by the external state management system.

Next we study the throughput and latency characteristics of the proposed load-balancing and elasticity mechanisms. Fig. 6a shows an example load-balancing scenario with fixed resources. It shows the *last 1-min* average data rate per node for a deployment with 3 reducer nodes and average queue length for one of the nodes. The system is initially imbalanced (due to random placement of small number of reducer nodes) but stable (i.e., $q_L \leq \tau_q^{high}$ for all nodes). At around 500s, we repeatedly emit a small subset of words causing further imbalance. The pending queue length for reducer 1 starts increasing beyond the threshold indicating that the incoming data rate is beyond its processing capacity and it initiates the load-balancing process with its neighbor, reducer 3, which in



(a) Load balancing example (fixed resources). (b) Throughput characteristics as a function of data-rate and scale-out. (c) Backup node latency.

Fig. 6. Throughput and Latency Characteristics for Load Balance and Scale-out.

turn transfers some of its load to reducer 2 and reaches a stable state around 700s. Note that the system may not reach stable state if the increase in data rate is beyond the cumulative capacity of the cluster, in which case a "scale out" operation will be performed. Fig. 6b shows the system response and throughput as a function of increasing data rate. We observe that for given set of resources, the achieved throughput initially increases with increase in the incoming data rate. However, as the system reaches its capacity, the observed throughput flattens out (as indicated in Fig. 6b). As a result, the pending queue length for the resources goes beyond the threshold and a load-balance or scale-out process is initiated on an idle or acquired resource which allows the system to catchup with the incoming data rate. We observe a reaction time for scale out to be around 1.5–3 seconds which includes both the detection latency as well as message replay and state restoration. While the former is a function of monitoring interval, the latter depends on both the checkpointing interval and the data rate (at the overloaded reducer), which we study next.

Fig. 6c shows the latency characteristics for the load-balancing scenario as a function of data rate and the checkpointing interval on the latency. We observe that the absolute value of the latency is very low (10 - 500ms) for moderate data rates and checkpointing interval of up to 10s. However, latency increases as the checkpointing interval and the data rate increases as this causes the number of tuples backed up by the node to increase along with the number of tuples that need to be replayed. Further, it adds significant memory overhead and that contributes to the performance degradation. Thus smaller checkpointing intervals are preferred. Another benefit of our approach is due to the proposed state representation and use of state fragments which allows us to asynchronously and consistently checkpoint a part of the state without pausing the regular operations, eliminating the effect of frequent write operations caused by small checkpointing intervals. As shown in Fig. 7, the proposed incremental checkpointing significantly

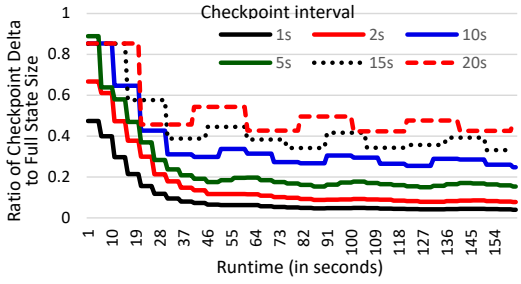
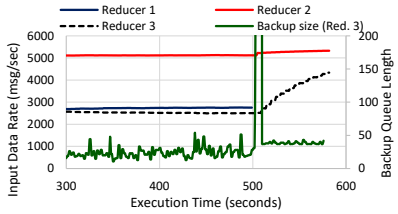
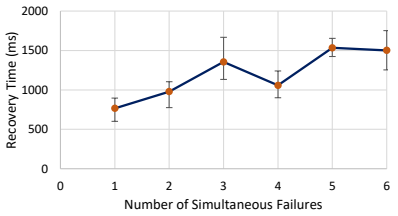


Fig. 7. Relative state size for incremental checkpointing.



(a) Fault-recovery example.



(b) Recovery latency as a function of number of simultaneous node failures (n = 12, r = 1, data rate = 15,000 msgs/sec).

Fig. 8. Fault Tolerance Throughput and Latency Characteristics.

reduces the size of the checkpoint when using smaller intervals (the checkpoint size for 2s interval stabilizes around 7% of the size of the entire state stored by the reducer node), which further supports our argument for a smaller checkpointing interval. Since load balancing, elasticity, and fault-tolerance use the same backup mechanism, Fig. 6c is a good indicator of the overall performance of our process.

Finally, we demonstrate the fault-tolerance and recovery process. Fig. 8a shows a snapshot of an execution with 3 reducer nodes (12 reducers) with fixed data rate. It also shows the size of the tuple backup for one of the reducers (reducer 3). We induce a fault in the system by manually stopping reducer 1 at around 500s. Reducer 3 stops receiving checkpoint data from the failed reducer (which is also treated as a heartbeat) leading to large tuple backup during the recovery process. After detecting the fault, it decides to take over the execution from the failed reducer and replays all the backed-up tuples to recover the state and finally moves itself on the ring so that the tuples originally destined for the failed node are now transmitted to reducer 3 (as is evident by the increasing data rate). Note that the latency characteristics of fault-recovery due to a *single node fault* are similar to that of the load-balancing process (Fig. 6c) and hence are omitted for brevity. We further study the recovery latency of the system under multiple concurrent failures. Since the recovery is performed in parallel, the overall recovery latency is the maximum of latencies to recover all the failed nodes in parallel. Figure 8b shows the average recovery latency observed for multiple (m)

simultaneous failures such that no two consecutive neighbors fail simultaneously. We observe that the recovery latency does not increase linearly and stabilizes around 1,200ms to 1,500ms for 3 to 6 simultaneous node failures. Note that the recovery latency for multiple simultaneous failures is measured as the maximum recovery latency incurred by any of the backup nodes in the ring. The observed variations in the recovery time (Fig. 8b) are due to the imbalances in the load (which leads to varying recovery latency for different failed nodes) and is not due to the increase in the number simultaneous failures.

IX. Conclusions

As high-velocity data becomes increasingly common, using the familiar MapReduce model to process it is valuable. In this paper, we presented an integrated approach to support fault-tolerance, load-balancing and elasticity for the streaming MapReduce model. Our novel approach extends the concept of consistent hashing, and provides locality-aware tuple peer-backup and peer-checkpointing. These allows low-latency dynamic updates to the system, including adaptive load-balance and elasticity, as well as offer low-latency fault recovery by eliminating the need for explicit state or tuple transfer during such operations. Our decentralized coordination mechanism helps make autonomic decisions based on the local information and eliminates a single point of failure. Our experiments show up to $2\times$ improvement in throughput compared to Storm SPS and demonstrated low-latency recovery of $10 - 1500\ ms$ from multiple concurrent failures. As future work, we plan to extend the evaluation to larger systems and real-world, long running application and also to extend the idea to a general purpose stream processing systems for wider applicability.

Acknowledgment

This work was supported by the U.S. National Science Foundation under grant ACI-1339756.

References

[1] W. Yin, Y. Simmhan, and V. Prasanna, "Scalable regression tree learning on hadoop using openplanet," in *MAPREDUCE*, 2012.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *HotCloud*, 2010.

[4] N. Parikh and N. Sundaresan, "Scalable and near real-time burst detection from ecommerce queries," in *SIGKDD*. ACM, 2008, pp. 972–980.

[5] A. Martin, C. Fetzer, and A. Brito, "Active replication at (almost) no cost," in *SRDS*. IEEE, 2011, pp. 21–30.

[6] M. Gatti, P. Cavalin, S. B. Neto, C. Pinhanez, C. dos Santos, D. Gribel, and A. P. Appel, "Large-scale multi-agent-based modeling and simulation of microblogging-based online social network," in *Multi-Agent-Based Simulation XIV*. Springer, 2014, pp. 17–33.

[7] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *ICDMW*, 2010.

[8] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer, "Scalable and low-latency data processing with stream mapreduce," in *CloudCom*. IEEE, 2011, pp. 48–58.

[9] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10.

[11] T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu, "Scalable parallel computing on clouds using twister4azure iterative mapreduce," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1035 – 1048, 2013.

[12] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen, "G-hadoop: Mapreduce across distributed data centers for data-intensive computing," *Future Generation Computer Systems*, vol. 29, no. 3, 2013.

[13] Q. Zheng, "Improving mapreduce fault tolerance in the cloud," in *International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, April 2010, pp. 1–6.

[14] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," *Computer Networks*, vol. 31, no. 11, 1999.

[15] M. F. Kaashoek and D. R. Karger, "Koorde: A simple degree-optimal distributed hash table," in *Peer-to-peer systems II*. Springer, 2003.

[16] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *Supercomputing*. ACM, 2004.

[17] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: cloud computing for everyone," in *Symposium on Cloud Computing*. ACM, 2011, p. 16.

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Workshop on MapReduce and its Applications (MAPREDUCE)*, 2010.

[19] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *Symposium on Cloud Computing*. ACM, 2011, p. 7.

[20] Storm, distributed and fault-tolerant realtime computation. Last accessed 19 Dec. 2014. [Online]. Available: http://storm.apache.org/

[21] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce," in *CLUSTER*. IEEE, 2009.

[22] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *ECCS*, 2013.

[23] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an elastic stream computing platform for the cloud," in *CLOUD*, 2011.

[24] A. Kumbhare, Y. Simmhan, and V. Prasanna, "Plasticc: Predictive look-ahead scheduling for continuous dataflows on clouds," in *CCGrid*. IEEE/ACM, May 2014, pp. 344–353.

[25] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *USENIX HotCloud*, 2012.

[26] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *SIGMOD*. ACM, 2004.

[27] M. Migliavacca, D. Eyers, J. Bacon, Y. Papagiannis, B. Shand, and P. Pietzuch, "Seep: scalable and elastic event processing," in *Middleware'10 Posters and Demos Track*. ACM, 2010, p. 4.

[28] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*. ACM, 2013, pp. 725–736.

[29] A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, C. Fetzer, and A. Brito, "Low-overhead fault tolerance for high-throughput data processing systems," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, June 2011, pp. 689–699.

[30] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal, "Extended virtual synchrony," in *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*. IEEE, 1994.

[31] C. Fetzer, "Streammine: A scalable and dependable event processing platform," in *DEBS*. ACM, 2010.

[32] Y. Simmhan, A. Kumbhare, and C. Wickramachari, "Floe: A dynamic, continuous dataflow framework for clouds," USC, Tech. Rep., 2013.

[33] A. Kumbhare, Y. Simmhan, and V. K. Prasanna, "Exploiting application dynamism and cloud elasticity for continuous dataflows," in *SuperComputing,*. ACM, 2013.

[34] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, 2007.

[35] J. S. Plank, J. Xu, and R. H. Netzer, "Compressed differences: An algorithm for fast incremental checkpointing," *University of Tennessee, Tech. Rep. CS-95-302*, 1995.

[36] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers," in *SuperComputing*. IEEE Computer Society, 2005.

[37] J. Dean, "Designs, lessons and advice from building large distributed systems," *Keynote from LADIS*, 2009.

APPENDIX

We include here additional content to clarify some of the core concepts outlined in the paper. Fig. 9 shows the streaming word frequency count application using stateful streaming map reduce (SMR) programming model. The Map function is executed for each incoming tuple (e.g. once per tweet), which then emits a tuple as a key value pair $\langle Word, 1\rangle$. Each tuple is then mapped to a reducer instance based on the specific key, similar to the batch mapreduce version. However, unlike batch MR, the reducer function is executed for each tuple it receives. Hence the reducer does not have access to all tuples for the corresponding key, instead the SMR framework maintains a local state associated with each "key" processed by the reducer (§V-C), which is passed along with the tuple to the reducer function. Hence the reducer keeps a running count of all the words seen so far. It can choose to emit the current count based on certain condition such as an external signal.

```
1: procedure MAP(Tuple, Emitter)
2:       → Tuple : Incoming data tuple
3:       → Emitter : Emit tuples to the reducers
4:    for Word in Tuple.value do
5:       Out ← ⟨Word, 1⟩
6:       Emitter.Write(Out)
7:    end for
8: end procedure
1: procedure REDUCE(Tuple, State, Emitter)
2:       → Tuple⟨k_i, v⟩ : Incoming tuple with key k_i
3:       → State : State associated with the key k_i
4:       → Emitter : Used to emit output tuples
5:    Word ← Tuple.key
6:    ct ← State.get("count")
7:    ct ← ct + Tuple.value
8:    State.Update("count", ct)
9:    if Condition then
10:      out ← ⟨Word, ct⟩
11:      Emitter.write(out)
12:   end if
13: end procedure
```

Fig. 9. Stateful Streaming MapReduce Word Count Example.

Fig. 10 shows a sample peer-ring configuration consisting of 4 reducer nodes. Assuming that the node (A) is overloaded, it negotiates with its neighbor (B) to check if it has spare cycles (i.e. $q_L \leq \tau_q^{low}$ *AND* $c \leq \tau_c^{low}$). If so, it requests B to take over some of its load (i.e. share the key space). Node B is then moved along the circle in the counter clockwise direction and its key-space is extended and hence it shares some load from node A. Note that during this process, no explicit state transfer is required si ad-balancing and hence ha ociated with the given ke

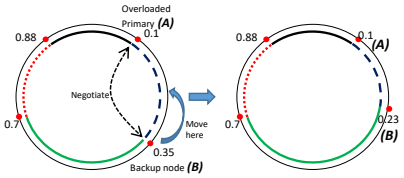


Fig. 10. Load balancing example.

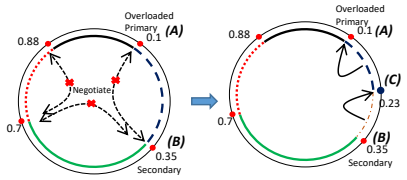


Fig. 11. Scaling out example

Fig. 11 shows a similar transition for scaling out. An overloaded node (A), first sends a load-balance request to its neighbor (B). However, unlike before, the neighbor (B) does not have enough free processing cycles and hence in turn tries to offload some load to its neighbor and so on. If it is observed that all the reducers are processing at its capacity, a new node C is provisioned and placed between A and B, which takes over some of the load from node A. However, note that in this case, node C does not have access to the state associated with the keys. Hence we perform a delayed transition (i.e. we start asynchronous state transfer to node C) while A continues to process the incoming tuple, and the load is transferred only after the state transfer is completed. Finally, Fig. 12 shows the transition for scaling in resources. In this case, since node A is lightly loaded, it checks with its neighbor node B if it has spare cycles to take over all of its load. If so, node A can be removed from the peer-ring and node B is moved into its position. As before, no state or tuple transfer is required to complete the scaling-in transition. Note that even though node B now proc other nodes, the low tupl ples without getting overl

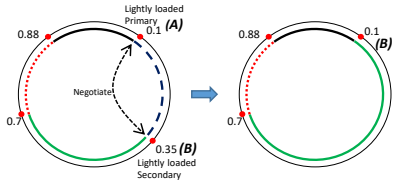


Fig. 12. Scaling in example.

Fig. 13 shows a peer-ring ($r = 1$) distributed across two fault zones. Given the property of fault-zones that the probability of simultaneous failures across two zones is much less than that of within a fault zone, the configuration with consecutive neighbors in the ring lying in distinct fault-zones as shown in fig. 13 gives an optimal fault-tolerance level. In the best case scenario, the system will still be operational without any state or tuple loss even if simultaneous failures are observed in *all* the nodes in a single fault zone (i.e. even if $\frac{n*1}{(1+1)} =$ of the nodes fail at the same time).

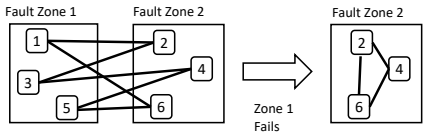


Fig. 13. Reducer peer ring with two fault zones: before and after failure.

Reactive Resource Provisioning Heuristics for Dynamic Dataflows on Cloud Infrastructure

Alok Gautam Kumbhare, *Student Member, IEEE,* Yogesh Simmhan, *Senior Member, IEEE,* Marc Frincu, *Member, IEEE,* and Viktor K. Prasanna, *Fellow, IEEE*

Abstract—The need for low latency analysis over high-velocity data streams motivates the need for distributed continuous dataflow systems. Contemporary stream processing systems use simple techniques to scale on elastic cloud resources to handle variable data rates. However, application QoS is also impacted by variability in resource performance exhibited by clouds and hence necessitates autonomic methods of provisioning elastic resources to support such applications on cloud infrastructure. We develop the concept of **"dynamic dataflows"** which utilize *alternate tasks* as additional control over the dataflow's cost and QoS. Further, we formalize an optimization problem to represent deployment and runtime resource provisioning that allows us to balance the application's *QoS*, *value*, and the resource *cost*. We propose two greedy heuristics, *centralized* and *sharded*, based on the variable-sized bin packing algorithm and compare against a *Genetic Algorithm (GA)* based heuristic that gives a near-optimal solution. A large-scale simulation study, using the Linear Road Benchmark and VM performance traces from the AWS public cloud, shows that while GA-based heuristic provides a better quality schedule, the greedy heuristics are more practical, and can intelligently utilize cloud elasticity to mitigate the effect of variability, both in input data rates and cloud resource performance, to meet the QoS of fast data applications.

Index Terms—Dataflows, stream processing, cloud, resource management, scheduling, high velocity data, runtime adaptation

1 Introduction

The expansion of ubiquitous virtual and physical sensors, leading up to the Internet of Things, has accelerated the rate and quantity of data being generated continuously. As a result, the need to manage and analyze such high *velocity* data in real-time forms one of the three dimensions of "Big Data", besides *volume* and *variety* [1]. While the past decade has seen sophisticated platforms for scalable offline analytics on large data volumes [2], Big Data systems for continuous analytics that adapt to the number, rate and variability of streams are relatively less well-studied.

There is a growing class of streaming applications in diverse domains: trend analysis and social network modeling for *online advertising* [3], real-time event processing to detect abnormal behavior in *complex systems* [4], and mission-critical use-cases such as *smart traffic* signaling [5] and demand-response in *smart power grids* [6]. These applications are characterized by the variety of input data streams, each with variable data rates. Further, data arrives at high velocity and needs to be analyzed with guaranteed low-latency even in the presence of data rate fluctuations. Hence, such applications lie at the intersection of the velocity and variety dimensions of the Big Data landscape.

While run-time scalability and seamless fault-tolerance together are the key requirements for handling high velocity variable-rate data streams, in this paper, we emphasize on *scalability* of stream processing applications with increasing data velocity and their *adaptation* to fluctuating data rates. Existing stream processing (or continuous dataflow) systems (SPS) [7], [8], [5], [9] allow users to compose applications as task graphs that consume and process continuous data, and execute on distributed commodity clusters and clouds. These systems support scalability with respect to high input data rates over static resource deployments, assuming the input rates are stable. When the input rates change, their static resource allocation causes over- or under-provisioning, resulting in wasted resources during low data rate periods and high processing latency during high data rate periods. Storm's *rebalance* [8] function allows users to monitor the incoming data rates and redeploy the application on-demand across a different set of resources, but requires the application to be paused. This can cause message loss or processing delays during the redeployment. As a result, such systems offer limited *self-manageability* to changing data rates, which we address in this article.

Recent SPSs such as Esc [10] and StreamCloud [11] have harnessed the cloud's elasticity to dynamically acquire and release resources based on application load. However, several works [12], [13], [14] have shown that the performance of public and private clouds themselves vary: for different resources, across the data center, and over time. Such variability impacts low latency streaming applications, and causes adaptation algorithms that assume reliable resource performance to fail. Hence, another gap we address here is autonomic *self-optimization* to respond to cloud performance variability for streaming applications. In addition, the pay-per-use cost model of commercial clouds requires intelligent resource management to minimize the real cost while satisfying the streaming application's quality of service (QoS) needs.

- *A. Kumbhare is with the Department of Computer Science, and V. K. Prasanna and M. Frincu are with Department of Electrical Engineering at University of Southern California, Los Angeles, USA. Email: {kumbhare, prasanna, frincu}@usc.edu*
- *Y. Simmhan is with Supercomputer, Education and Research Centre (SERC) at Indian Institute of Science, Bangalore, India. Email: simmhan@serc.iisc.in*

In this article we push towards autonomic provisioning of continuous dataflow applications to enable scalable execution on clouds by leveraging cloud elasticity and addressing the following issues:

1) Autonomous runtime adaptions in response to fluctuations in both input data rates and cloud resource performance.
2) Offering flexible trade-offs to balance monetary cost of cloud resources against the users' perceived application value.

This article extends our previous work [15], which introduced the notion of "dynamic dataflows" and proposed greedy reactive resource provisioning heuristics (§ 8) to exploit cloud elasticity and the flexibility offered by dynamic dataflows. Our contributions in this article are:

- **Application Model and Optimization Problem:** We develop the application model for **dynamic dataflows** (§ 2) as well as the infrastructure model to represent IaaS cloud characteristics (§ 3), and propose an optimization problem (§ 6) for resource provisioning that balances the resource cost, application throughput and the domain value based on user-defined constraints.
- **Algorithms:** We present a *Genetic Algorithm (GA)*-based heuristic for deployment and runtime adaptation of continuous dataflows (§ 7) to solve the optimization problem. We also propose efficient greedy heuristics (*centralized* and *sharded* variants) that sacrifice optimality over efficiency, which is critical for low latency streaming applications (§ 8).
- **Evaluation:** We extend the *Linear Road Benchmark (LRB)* [16] as a **dynamic dataflow** application, which incorporates dynamic processing elements, to evaluate the reactive heuristics through large-scale simulations of LRB, scaling up to 8,000 msgs/sec, using VM and network performance traces from Amazon AWS cloud service provider. Finally, we offer a *comparative analysis* of the greedy heuristics against the GA in terms of scalability, profit, and QoS (§ 9.2).

2 Dynamic Dataflow Application Model

We leverage the familiar Directed Acyclic Graph (DAG) model to define *Continuous Dataflows* (Def. 1). This allows users to compose loosely coupled applications from individual tasks with data dependencies between them defined as streaming dataflow edges. While, in practice, this model can be extended to include more complex constructs like back flows/cycles, we limit our discussion to DAGs to keep the application model simple and the optimization problem tractable.

Def. 1: A continuous dataflow G is a quadruple $G = \langle P, E, I, O \rangle$, where $P = \{P_1, P_2, ..., P_n\}$ is the set of *Processing Elements (PE)* and $E = \{\langle P_i, P_j \rangle \mid P_i, P_j \in P\}$ is a set of directed *dataflow* edges without cycles such that data *messages* flow from P_i to P_j. $I \neq \varnothing \subset P$ is a set of *input PEs* which receive messages only from external data streams, and $O \neq \varnothing \subset P$ is a set of *output PEs* that emit output messages only to external entities.

Each PE represents a long-running, user-defined task which executes continuously, accepting and consuming messages from its incoming ports and producing messages on the outgoing ports. A directed edge between two PEs connects an output port from the source PE to an input port of the sink PE, and represents a flow of messages between the two. Without loss of generality, we assume *and-split semantics* for edges originating from the same output port of a PE (i.e., output messages on a port are duplicated on all outgoing edges) and *multi-merge semantics* [17] for edges terminating at an input port of another PE (i.e., input messages from all incoming edges on a port are interleaved).

We define *Dynamic Dataflows* (Def. 2) as an extension to continuous dataflows by incorporating the concept of dynamic PEs [15]. Dynamic PEs consists of one or more user-defined alternative implementations *(alternates)* for the given PE, any one of which may be selected as an active alternate at run-time. Each alternate may possess different *performance characteristics*, *resource requirements* and *domain perceived functional quality (value)*. Heterogeneous computing [18] and (batch processing) workflows [19] incorporate a similar notion where the active alternates are *decided once at deployment time* but thereafter remain fixed during execution. We extend this to continuous dataflows where alternate selection is an *on-going process at runtime*. This allows the execution framework to perform autonomic adaptations by dynamically altering the active alternates for an application to meet its QoS needs based on current conditions.

Def. 2 (Dynamic Dataflow): A Dynamic Dataflow $D = \langle \mathcal{P}, E, \mathcal{I}, \mathcal{O} \rangle$ is a continuous dataflow where each PE $\mathcal{P}_i \in \mathcal{P}$ has a set of alternates $\mathcal{P}_i = \{p_i^1, p_i^2, ...p_i^j \mid j \geq 1\}$ where $p_i^j = \langle \gamma_i^j, c_i^j, s_i^j \rangle$. γ_i^j, c_i^j, and s_i^j denote the *relative value*, the *processing cost* per message, and the *selectivity* for the alternate p_i^j of PE $\mathcal{P}_i$ respectively.

Selectivity, s_i^j, is the ratio of the number of output messages produced to the number of input messages consumed by the alternate p_i^j to complete a logical unit of operation. It helps determine the outgoing data rate of a PE relative to its input data rate, and thereby its cascading impact on downstream PEs in the dataflow.

Each alternate has associated cost and value functions to assist with alternate selection and resource provisioning decisions. The *relative value*, $0 < \gamma_i^j \leq 1$, for an alternate p_i^j is:

$$\gamma_i^j = \frac{f(p_i^j)}{\text{MAX}_j\{f(p_i^j)\}} \tag{1}$$

where $f : \mathcal{P}_i \to \mathbb{R}$ is a user-defined *value function* for the alternates. It quantifies the relative domain benefit to the user of picking that alternate. For e.g., a Classification PE that classifies its input tuples into different classes may use the F_1 score[1] as the quality of that algorithm to the domain, and F_1 can be used to calculate the *relative value* for alternates of the PE.

1. $F_1 = 2 \times \frac{precision \times recall}{precision + recall}$ is a measure of the classifier's labeling accuracy.

```
<dataflow>
 <PE id="parser">
  <in id="tweet" type="string" />
  <out id="cleaned" type="string" />
  <alternate impl="Parser.TweetParser" />
 </PE>
 <PE id="classifier">
  <in id="cleaned_twt" type="string" />
  <out id="tpc_twt" type="string" />
  <alternate impl="Classifier.Bayes" value="0.65" />
  <alternate impl="Classifier.LDA" value="0.70" />
  <alternate impl="Classifier.MWE" value="1.00" />
 </PE>

 <edge source="parser:cleaned"
       sink="classifier:cleaned_twt" />
</dataflow>
```

Fig. 1: Sample declarative representation of Dynamic Dataflow using XML. Equivalent visual representation is in Fig. 2(a).

Finally, the *processing cost* per message, c_i^j, is the time (in seconds) required to process one message on a "reference" CPU core (§ 3) for the alternate p_i^j. The processing needs of an alternate determines the resources required to processes incoming data streams at a desired rate.

The concept of dynamic PEs and *alternates* provides a powerful abstraction and an additional point of control to the user. A sample dynamic dataflow is shown in Fig. 1 using a generic XML representation, with a visual equivalent shown in Fig. 2(a). Any existing dataflow representation, such as declarative [20], functional [8] or imperative [9] may also be used. The sample dataflow continuously parses incoming tweets, and classifies them into different topics. It consists of two PEs: *parser*, and *classifier*, connected using a dataflow edge. While the *parser* PE consists of only one implementation, the *classifier* PE consists of three alternates, using the Bayes, Latent Dirichlet Allocation (LDA) and Multi-Word enhancement (MWE) to LDA algorithms, respectively. Each alternate varies in classification accuracy and hence has different *value* to the domain; these are normalized relative to the best among the three. The three alternates are available for dynamic selection at runtime. For brevity, we omit a more detailed discussion of the dynamic dataflow programming abstraction.

The execution and scalability of a dynamic dataflow application depends on the capabilities of the underlying infrastructure. Hence, we develop an infrastructure model to abstract the characteristics that impacts the application execution and use that to define the resource provisioning optimization problem (§ 6).

3 Cloud Infrastructure Model

We assume an *Infrastructure as a Service (IaaS)* cloud that provides access to virtual machines (VMs) and a shared network. In IaaS clouds, a user has no control over the VM placement on physical hardware, the multi-tenancy, or the network behavior between VMs. The cloud environment provides a set of *VM resource classes* $\mathcal{C} = \{C_1, C_2, ..., C_n\}$ that differ in the number of available virtual CPU cores N, their *rated core speed* π, and their *rated network bandwidth* β. In this article, we focus on CPU bound PEs that operate on incoming data streams from the network. As a result, we ignore memory and disk characteristics and only use the VM's CPU and network behavior in the infrastructure performance model used by our adaptation heuristics.

As CPU core speeds may vary across VM classes, we define the *normalized processing power* π_i of a resource class C_i's CPU core as the ratio of its processing power to that of a *reference* VM core. Naïvely, this may be the ratio of their clock speeds, but could also be obtained by running application benchmarks on different sets of VMs and comparing them against a defined reference VM, or use Cloud-providers' "ratings" such as Amazon's Elastic Compute Units (ECUs).

The set of *VM resources acquired* till time t is denoted by $R(t) = \{r_1, r_2, ..., r_n\}$. Each VM is described by $r_i = \langle C_j^i, t_{start}^i, t_{stop}^i \rangle$ where C_j^i is the resource class to which the VM belongs, and t_{start}^i and t_{stop}^i are the times at which the VM was acquired and released, respectively. $t_{stop} = \infty$ for an active VM.

The peer-to-peer network characteristic between pairs of VMs, r_i and r_j, is given by $\lambda_{i\times j}$ and $\beta_{i\times j}$, where $\lambda_{i\times j}$ is the *network latency* between VM r_i and r_j and $\beta_{i\times j}$ is their *available bandwidth*.

VMs are typically charged at whole VM-hours by current cloud providers. The user is billed for the entire hour even if a VM is released before an hour boundary. The *total accumulated monetary cost* for the VM r_i at time t is then calculated as:

$$\mu_i(t) = \lceil \frac{\min(t_{stop}, t) \; - \; t_{start}}{60} \rceil \times cost \; per \; VM \; hour \quad (2)$$

where $\min(t_{stop}, t) - t_{start}$ is the duration in minutes for which the VM has been active.

We gauge the on-going performance of virtualized cloud resources, and the variability relative to their rated capability, using a presumed monitoring framework. This periodically probes the compute and network performance of VMs using standard benchmarks. The *normalized processing power* of a VM r_i observed at time t is given by $\pi_i(t)$, and the *network latency and bandwidth* between pairs of active VM r_i and r_j are $\lambda_{i\times j}(t)$ and $\beta_{i\times j}(t)$, respectively. To minimize overhead, we only monitor the network characteristics between VMs that host neighboring PEs in the DAG to assess their impact on dataflow throughput. We assume that rated network performance as defined by the provider is maintained for other VM pairs. Two PEs collocated in the same VM are assumed to transfer messages in-memory, i.e., $\lambda_{i\times i} \to 0$ and $\beta_{i\times i} \to \infty$.

4 Deployment and Adaptation Approach

Based on the dynamic dataflow and cloud infrastructure models, we propose a deployment and autonomic runtime adaptation approach that attempts to balance *simplicity*, realistic *cloud characteristics* (e.g., billing model, elasticity), and *user flexibility* (e.g., dynamic PEs). Later, we formally define a meaningful yet tractable optimization problem for the deployment and runtime adaptation strategies (§ 6).

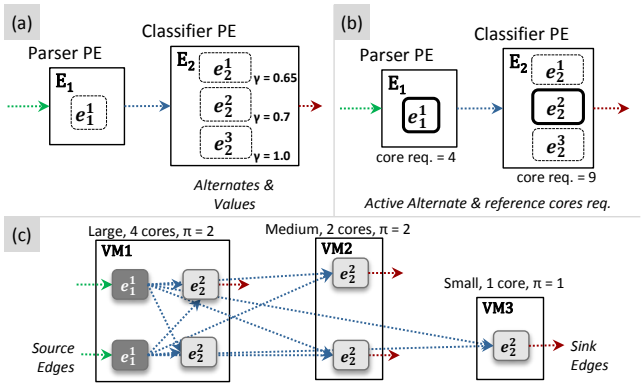


Fig. 2: (a) A sample dynamic dataflow. (b) Dataflow with selected alternates (e_1^1, e_2^2) and their initial core requirements. (c) A deployment of the dataflow onto VMs.

We make several practical assumptions on the continuous dataflow processing framework, reflecting features available in existing systems [21], [8]:

1) The dataflow application is deployed on distributed machines, with a *set of instances of each PE* ($\varphi(t)$ at time t) running in parallel across them. Incoming messages are actively load-balanced across the different PE instances based on the processing power of the CPU cores they run on and the length of the pending input queue. This allows us to easily scale the application by increasing the number of PE instances if the incoming load increases.
2) Within a single multi-core machine, multiple instances of different PEs run in parallel, isolated on separate cores. A core is exclusively allocated to one PE instance. We assume that there is minimal interference from system processes.
3) Running n data-parallel instances of a PE on n CPU cores, each with *processing power* $\pi = 1$, is equivalent to running 1 instance of the PE on 1 CPU core with $\pi = n$.
4) The active alternate for a PE is not dependent on the active alternate of any other PE, since each alternate for a given PE follows the same input/output format. This allows us to independently switch the active alternate for different PEs during runtime.
5) The framework can spin up or shutdown cloud VMs on-demand (with an associated startup/shutdown latency) and can periodically monitor the VM characteristics, such as CPU performance and network bandwidth.

Given these assumptions we define the following deployment model. When a dynamic dataflow, such as Fig. 2(a), is submitted for execution, the *scheduler* for the stream processing framework needs to make several decisions: alternate selection for each dynamic PE, acquisition of VMs, mapping of these PEs to the acquired VMs, and deciding the number of data parallel instances per PE. These activities are divided into two phases: *deployment time* and *runtime* strategies.

Deployment time strategies select the initial active alternate for each PE, and determine their CPU core requirements (relative to the "reference" core) based on *estimated* initial message data rates and rated VM performance. Fig. 2(b) shows the outcome of selecting alternates, picking e_1^1 and e_2^2 for PEs E_1 and E_2 with their respective core requirements. Further, it determines the VMs of particular resource classes that are instantiated, and the mapping $\mathcal{M}$ from the data-parallel instances of the each PE $(\varphi(t))$ to the active VMs $(R(t))$, following which the dataflow execution starts. Fig. 2(c) shows multiple data-parallel instances of these PEs deployed on a set VMs of different types. Note that the number of PE instances in Fig. 2(c) – $\varphi(t)$ is 2 and 5 for e_1^1 and e_2^2 – is not equal to the core requirements in Fig. 2(b) – 4 and 9 cores – since some instances are run on faster CPU cores ($\pi > 1$).

Runtime strategies, on the other hand, are responsible for periodic adaptations to the application deployment in response to the variability in the input data rates and the VM performance obtained from the monitoring framework. These active decisions are determined by the runtime heuristics that can decide to switch the active alternate for a PE, or change the resources allocated to a PE within or across VMs. The acquisition and release of VMs are also tied to these decisions as they determine the actual cost paid to the cloud service provider. A formal definition of the optimization problem and these control strategies will be presented in § 6.

5 Metrics for Quality of Service

In § 2, we captured the value, and processing requirements for individual PEs and their alternates using the metrics: *relative value* (γ_i^j), *alternate processing cost* (c_i^j), and *selectivity* (s_i^j). In this section, we expand these QoS metrics to the entire dataflow application.

We define an *optimization period* T for which the dataflow is executed. This optimization period is divided into *time intervals* $T = \{t_0, t_1, ..., t_n\}$. We assume these *interval lengths* are constant, $\triangle t = t_{i+1} - t_i$. For brevity we omit the suffix i while referring to the time interval t_i, unless necessary for disambiguation.

A dataflow is initially deployed with a particular configuration of alternates which can later be switched during runtime to meet the application's QoS. To keep the problem tractable and avoid repetitive switches, these changes are only made at the start of each interval t_i. This means that during a time interval t, only a specific alternate for a PE $\mathcal{P}_i$ is active. The *value* of the PE $\mathcal{P}_i$ during the time interval t is thus:

$$\Gamma_i(t) = \sum_{p_i^j \in P_i} \left(A_i^j(t) \cdot \gamma_i^j \right)$$

$$A_i^j(t) = \begin{cases} 1, & \text{if alternate } p_i^j \text{ is active at time } t \\ 0, & \text{otherwise} \end{cases}$$

Since value can be perceived as an additive property [22] over the dataflow DAG, we aggregate the individual values of active alternates to obtain the value for the entire dynamic dataflow during the time interval t.

Def. 3 (Normalized Application Value): The *normalized application value*, $0 < \Gamma(t) \leq 1$, for a dynamic dataflow D during the time interval t is:

$$\Gamma(t) = \frac{\sum_{\mathcal{P}_i \in \mathcal{P}} \Gamma_i(t)}{|\mathcal{P}|} \tag{3}$$

where $|\mathcal{P}|$ is the number of PEs in the dataflow.

The application's value thus obtained gives an indication of its overall quality from the domain's perspective and can be considered as one QoS dimension.

Another QoS criterion, particularly in the context of continuous dataflows, is the observed application throughput. However, raw application throughput is not meaningful because it is a function of the input data rates during that time interval. Instead we define the *relative application throughput*, Ω, built up from the *relative throughput* $0 < \Omega_i(t) \leq 1$ of individual PEs $\mathcal{P}_i$ during the interval t. These are defined as the ratio of the PEs' current output data rate (absolute throughput) $o_i(t)$ to the maximum achievable output data rate $o_i^{max}(t)$: $\Omega_i(t) = \frac{o_i(t)}{o_i^{max}(t)}$

The output data rate for a PE depends on the selectivity of the active alternate, and is bound by the total resources available to the PE to process the inputs given the processing cost of the active alternate. The actual *output data rate* during the interval t is:

$$o_i(t) = \frac{\min\left(q_i(t) + i_i(t) \cdot \Delta t, \quad \frac{\phi_i \cdot \Delta t}{c_i^j}\right) \times s_i^j}{\Delta t} \tag{4}$$

where $q_i(t)$ is the *number of pending input messages* in the queue for PE $\mathcal{P}_i$, $i_i(t)$ is the *input data rate* in the time interval t for the PE , ϕ_i is its *total core allocation* for the PE ($\phi_i = \sum_k \pi_k$), and c_i^j is the processing cost per message and s_i^j is the selectivity for the active alternate p_i^j.

The *maximum output throughput* is achieved when there are enough resources for $\mathcal{P}_i$ to process all incoming data messages including messages pending in the queue at the start of the interval t. This is given by $o_i^{max}(t) = \frac{(q_i(t)+i_i(t)\times\Delta t)\times s_i^j}{\Delta t}$.

While the input data rate for the source PE is determined externally, the input rate for other PEs can be characterized as follows. The flow of messages between consecutive PEs is limited by the bandwidth, during the interval t, between the VMs on which the PEs are deployed. We define the *flow* $f_{i,j}(t)$ from $\mathcal{P}_i$ to $\mathcal{P}_j$ as:

$$f_{i,j}(t) = \begin{cases} \min\left(o_i, \quad \frac{\beta_{i,j}(t)\cdot\triangle t}{m}\right), & \langle \mathcal{P}_i, \mathcal{P}_j \rangle \in E \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

where $\beta_{i,j}(t)$ is the available cumulative bandwidth between all instances of $\mathcal{P}_i$ and $\mathcal{P}_j$ at time t and m is the *average output message size*.

Given the *multi-merge semantics* for incoming edges, the input data rate for $\mathcal{P}_k$ during time t is:

$$i_k(t) = \sum f_{j,k}(t) \tag{6}$$

Unlike the application's value, its relative throughput is not additive as it depends on the critical processing path of the PEs during that interval. The relative throughput for the entire application is the ratio of observed cumulative outgoing data rate from the output PEs, $\mathcal{O} = \{\mathcal{O}_i\}$, to the maximum achievable output rate for those output PEs, for the current input data rate.

Def. 4 (Relative Application Throughput): The *relative application throughput*, $0 < \Omega(t) \leq 1$, for dynamic dataflow $D = \langle \mathcal{P}, E, \mathcal{I}, \mathcal{O} \rangle$ during the time interval t is:

$$\Omega(t) = \frac{\sum_{\mathcal{P}_i \in \mathcal{O}} \Omega_i(t)}{|\mathcal{O}|} \tag{7}$$

The output data rate for the output PEs is obtained by calculating the output data rate of individual PEs (Eqn. 4) followed by the flow $f_{i,j}$ between consecutive PEs (Eqn. 5), repeatedly in a breadth-first scan of the DAG starting from its input PEs, till the input and output data rates for the output PEs is obtained.

Normalized Application Value, $\Gamma(t)$, and *Relative Application Throughput*, $\Omega(t)$, together provide complimentary QoS metrics to assess overall application execution, which we use to define an optimization problem that balances these QoS metrics based on user-defined constraints in the next section.

6 Problem Formulation

We formulate the optimization problem as a *constrained utility maximization* problem during the period T for which the dataflow is executed. The *constraint* ensures that the expected relative application throughput meets a threshold, $\Omega \geq \widehat{\Omega}$; the *utility* to be *maximized* is a function of the normalized application value, Γ; and the cost for cloud resources, μ, during the optimization period T.

For a dynamic dataflow $D = \langle \mathcal{P}, E, \mathcal{I}, \mathcal{O} \rangle$, the *estimated* input data rates, $I(t_0) = \{i_j(t_0)\}$, at each input PE, $\mathcal{P}_j \in \mathcal{I}$, at initial time, t_0, is given. During each subsequent time interval, t_i, based on the observations of the monitoring framework during t_{i-1}, we have the following: the *observed* input data rates, $I(t) = \{i_j(t)\}$; the set of active VMs, $R(t) = \{r_1, r_2, ..., r_m\}$; the normalized processing power per core for each VM r_j, $\pi(t) = \{\pi_j(t)\}$; the network latency and the bandwidth between pairs of VMs $r_i, r_j \in R(t)$ hosting neighboring PEs are $\lambda(t) = \{\lambda_{i\times j}(t)\}$ and $\beta(t) = \{\beta_{i\times j}(t)\}$, respectively.

At any time interval t, we can calculate the *relative application throughput* $\Omega(t)$ (Eqn. 7), the *normalized application value* $\Gamma(t)$ (Eqn. 3), and the cumulative monetary cost $\mu(t)$ till time t (Eqn. 2).

The *average* relative application throughput (Ω), the *average* relative application value (Γ), and the *total* resource cost (μ) for the entire optimization period $T = \{t_0, t_1, \ldots, t_n\}$ are:

$$\Omega = \frac{\sum_{t\in T} \Omega(t)}{|T|} \qquad \Gamma = \frac{\sum_{t\in T} \Gamma(t)}{|T|} \qquad \mu = \mu(t_n)$$

We define the combined *utility* as a function of both the total resource cost (μ) and the average application value (Γ). To help the users to trade-off between cost and value, we allow them to define the expected *maximum cost* at which they break-even for the two extremes of application value, i.e., the values obtained by selecting the *best alternates* for all PEs, on one end, and by selecting the *worst alternates*, on the other. For simplicity, we assume a linear function to derive the expected maximum resource cost at an intermediate application value, as

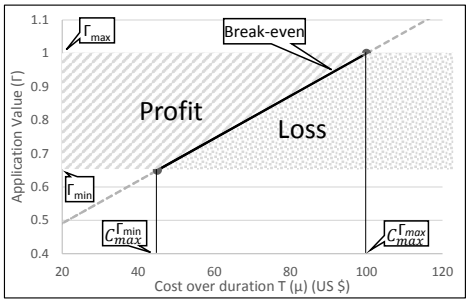


Fig. 3: A sample linear function for trade-off between cost (C) and value (Γ). The line denotes break-even, with a slope = σ.

shown in Fig. 3. If the actual resource cost lies below this break-even we consider it as profit, while if it lies above, we consider it as loss. This can be captured using the following *objective function*, Θ , which is to be maximized over the optimization period under the constraint $\Omega \geq \widehat{\Omega}$:

$$\Theta = \Gamma - \sigma \cdot (\mu - C_{max}^{\Gamma_{max}}) \quad (8)$$

where σ is a *equivalence coefficient* between cost and value given by the slope:

$$\sigma = \frac{\Gamma_{max} - \Gamma_{min}}{C_{max}^{\Gamma_{max}} - C_{max}^{\Gamma_{min}}} \quad (9)$$

Γ_{max} and Γ_{min} are the maximum and minimum possible relative application values when picking the alternates with the best and worst values for each PE, respectively, while $C_{max}^{\Gamma_{max}}$ and $C_{max}^{\Gamma_{min}}$ are the user-defined break-even resource cost at Γ_{max} and Γ_{min}.

Given the deployment approach (§ 4), the above objective function Θ can be maximized by choosing appropriate values for the following *control parameters* at the start of each interval t_i during the optimization period:

- $A_i^j(t)$, the active alternate j for the PE P_i
- $R(t) = \{r_j(t)\}$, the set of VMs in $R(t)$
- $\varphi(t) = \{\varphi_j(t)\}$, the set of data-parallel instances for each PE P_j; and
- $\mathcal{M}(t) = \{\varphi_j(t) \rightarrow \mathcal{M}_{j \times k}(t)\}$, the mapping of a data-parallel instances φ_j for PE P_j to the actual VM r_k

Optimally solving the objective function Θ with the Ω constraint is *NP-Hard*. The proof is outside the scope of this article and a sketch is presented in our earlier work [15]. While techniques like integer programming and branch-and-bound have been used to optimally solve some NP-hard problems [23], these do not adequately translate to low-latency solutions for continuous adaptation decisions. The dynamic nature of the application and the infrastructure, as well as the tightly-bound decision making interval means that fast heuristics performed repeatedly are better than slow optimal solutions. We thus propose simplified heuristics to provide an approximate solution to the objective function.

Other approximate procedures such as gradient descent are not directly applicable to the problem at hand since the optimization problem presents a non-differentiable, non-continuous function. However, nature-inspired search algorithms such as Genetic Algorithms (GAs), ant-colony optimization, and particle-swarm optimization, which follow a guided randomized

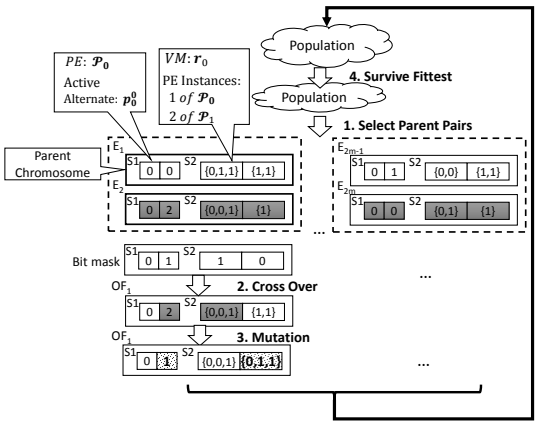


Fig. 4: Sample iteration for the GA heuristic.

search, are sometimes more effective than traditional heuristics in solving similar single and multi-objective workflow scheduling problems [24], [25], [26], [27]. In this article, we also explore GAs for finding an approximate solution to the optimization problem. While GAs are usually slow and depend on the population size and complexity of the operators used, they provide a good baseline to compare against our greedy heuristics and are discussed in the next section, followed by the greedy deployment and adaptation heuristics in section 8.

7 Genetic Algorithm-based Scheduling

A GA [28] is a meta-heuristic used in optimization and combinatorial problems. It facilitates the exploration of a large search space by iteratively evolving a number of candidate solutions towards the global optimum. The GA meta-heuristic abstracts out the structure of the solution for a given problem in terms of a *chromosome* made up of several *genes*. It then explores the solution space by evolving a set of chromosomes (potential solutions) over a number of generations.

The GA initially generates a random population of chromosomes which act as the seed for the search. The algorithm then performs genetic operations such as *crossover* and *mutations* to iteratively obtain successive generations of these chromosomes. The *crossover* operator takes a pair of parent chromosomes and generates an *offspring* chromosome by crossing over individual genes from each parent. This helps potentially combine partial solutions from the parents into a single offspring. Further, the *mutation* operator is used to randomly alter some parts of a given chromosomes and advance the search by possibly avoiding getting stuck in a local optimum. The algorithm then applies a *selection* operator which picks the best chromosomes from the entire population based on their *fitness values* and eliminates the rest. This process is repeated until a *stopping criterion*, such as a certain number of iterations or the convergence of a fitness value, is met.

We adapt GA to our optimization problem by defining these domain-specific data structures and operators.

Chromosome: The solution to the optimization problem (Eqn. 8) requires (1) determining the best alternate to activate for each PE, and (2) the type and number of

VMs, and the mapping of data-parallel PE instances to these VMs. We capture both of these aspects of the deployment configuration using a double stranded *chromosome* (Fig. 4). Each chromosome represents one possible deployment configuration and the goal is to find the chromosome with the optimal configuration. The first strand (S1) represents the PEs and their active alternates, with each gene in the strand representing a PE $\mathcal{P}_i$ in the dataflow The value of the i^{th} gene holds the index j of the active alternate p_i^j for the corresponding PE. The second strand (S2) contains genes that represent the list of available VMs with the list of PE instances running on them. The value of the i^{th} gene in strand S2 holds the list of index values j for PEs $\mathcal{P}_j$ mapped to the VM r_i. The chromosome size (i.e. number of genes) is fixed based on the available resource budget.

For example, in Fig 4, the chromosome E_1 represents a deployment where strand S1 has two PEs $\mathcal{P}_0$ and $\mathcal{P}_1$ with active alternates p_0^0 and p_1^0, respectively. Strand S2 identifies two VMs, r_0 and r_1, with r_0 running one instance of PE $\mathcal{P}_0$ and two instances of PE $\mathcal{P}_1$ on it, while r_1 has two instances of PE $\mathcal{P}_1$ running on it.

Fitness Function: The objective function Θ acts as the fitness function for the chromosome, and depends on both the strands of the chromosome. The throughput constraint $\Omega \geq \widehat{\Omega}$ is incorporated using a weighted *penalty function* that penalizes the fitness value if a constraint is violated. This ensures that while the chromosome is penalized, it is not disregarded immediately and given a chance to recover through mutations. In addition, during runtime, the penalty function also considers the number of changes induced on the deployment to reduce the overhead of frequent changes in the system.

$$penalty = \begin{cases} -\alpha \cdot \mid \Omega - \widehat{\Omega} \mid \cdot it & \text{, deployment} \\ -\alpha \cdot \mid \Omega - \widehat{\Omega} \mid \cdot it + \alpha' \cdot v \cdot it & \text{, runtime} \end{cases}$$

where, α and α' are constants, v is the number of deployment changes observed in the chromosome as compared to the previous deployment and it is the iteration count for GA. it ensures that the penalty is increased as the chromosome survives over multiple iterations and hence allows removal of unfit chromosomes.

Crossover: Each parent chromosome is first selected from the population using a probabilistic ranked model (similar to the Roulette wheel) [29] while also retaining the top 5% of the chromosomes with best fitness values. Next the parent chromosomes are paired randomly and a random bit mask is generated to choose the a gene from either parent to produce the offspring chromosome. For e.g., in Fig. 4, parents E_1 (white) and E_2 (gray) are selected for crossover, and a random bit mask is used to decide if the gene from the first parent (bit is 0) or the second parent (bit is 1) is retained in the offspring OF_1.

Mutation: We allow independent mutation for the two chromosome strands. For the first strand with PE alternates the mutation involves randomly switching the active alternate. For the second strand of VM instances and mapping, we probabilistically decide whether to remove or add a PE instance for each VM. For example, in Fig. 4, the offspring OF_1 undergoes mutation by switching the active alternate for $\mathcal{P}_1$ from $p_1^2 \rightarrow p_1^1$, and by adding an instance of $\mathcal{P}_0$ to the second VM. Mutated genes are shown with a dotted pattern.

While GAs explore a wide range of solutions and tend to give near-optimal solutions, their convergence performance becomes a bottleneck during runtime adaptation. Hence we design sub-optimal greedy heuristics that trade optimality for speed, making them better suited for streaming applications.

8 Greedy Deployment & Adaptation Heuristics

In this section, we propose greedy heuristics to find an approximate solution to the optimization problem. As before, the algorithm is divided into the initial deployment phase and the runtime adaptation phase.

For the proposed heuristic, we provide *sharded* (SH) and *centralized* (CE) variants that differ in the quanta of information needed and the execution pattern of the scheduler. The *sharded* version uses one scheduler per PE, and all data-parallel instances of a PE communicate with their scheduler, potentially across VMs. However, schedulers for different PEs do not communicate. Hence each scheduler only has access to its PE instances. In the *centralized* version, a single scheduler gathers information about the entire dataflow and hence has a global view of the execution of all PEs. As we show (§ 9.2), while the SH scheduler is inherently decentralized and reduces the transfer of monitoring data during execution, the CE variant, due to its global view, is more responsive to changes in the execution environment.

8.1 Initial Deployment Heuristic

The initial deployment algorithm (Alg. 1) is divided into two stages: Alternate selection (lines 2–11) and Resource allocation (lines 12–25). These algorithms are identical for both SH and CE schedulers; however, their costing functions differ (Table 1).

The alternate selection stage ranks each PE alternate based on the ratio of its value to estimated cost (line 4), and chooses the one with the highest ratio. Since we do not know the actual cost for the alternates until resource allocation, the heuristic uses the estimated processing requirements (c_P^A) as an approximation. The GetCostOfAlternate function varies between the SH and CE versions. The SH strategy calculates an alternate's cost based on only its processing requirements, while the CE strategy calculates the cost of the alternate as the sum of both its own processing needs and that of its downstream PEs – intuitively, if an upstream PE has more resources allocated, its output message rate increases and this has a cascading impact on the input rate of the succeeding PEs. Also, a higher selectivity upstream PE will further impact the successors' cost since they will have to process more messages. This cost is calculated using a dynamic programming algorithm by traversing

Algorithm 1 Initial Deployment Heuristic Algorithm

```
1: procedure INITIALDEPLOYMENT(Dataflow D)
                                          ▷ Alternate Selection Stage
2:     for PE P ∈ D do
3:         for Alternate A ∈ P do
4:             c_P^A ← GETCOSTOFALTERNATE(A)
5:             γ ← A.Value
6:             if γ/c_P^A ≥ best then
7:                 best ← γ/c_P^A
8:                 selected ← A
9:             end if
10:        end for
11:    end for
                                          ▷ Resource Allocation Stage
12:    while Ω ≤ Ω̂ do
13:        if (VM.isAvailable = false) then
14:            VM ← INITIALIZEVM(LargestVMClass)
15:        end if
16:        P ← GETNEXTPE
17:        CurrentAlloc ← ALLOCATENEWCORE(P, VM)
18:        Ω ← GETESTIMATEDTHRUPUT(D,CurrentAlloc)
19:    end while
20:    for PE P ∈ D do
21:        if ISOVERPROVISIONED(P) then
22:            REPACKPEINSTANCES(P)  ▷ Move PE instances to a VM
    with lower core capacity
23:        end if
24:    end for
25:    REPACKFREEVMS  ▷ Repack PEs in VMs with free cores to VMs
    with less number of cores
26: end procedure
```

the dataflow graph in reverse BFS order rooted at the output PEs.

This is followed by a resource selection stage (lines 12–25) which operates similar to the *variable-sized bin packing (VBP)* problem [30]. For the initial deployment, in the absence of running VMs, we assume that each VM from a resource class behaves ideally as per its rated performance. The algorithm picks PEs (objects) in an order given by GETNEXTPE, and puts an instance of each PE in the largest VM (bin) (line 17), creating a new VM (bin) if required. It then calculates the estimated relative throughput for the application given the current allocation (line 18) using Eqn. 7 and repeats the procedure if the application constraint is not met. It should be noted that the GETESTIMATEDTHROUGHPUT function considers both the allocated cores and the available bandwidth to calculate the relative throughput and hence scales out when either becomes a bottleneck.

The intuition behind GETNEXTPE is to choose PEs in an order that not only increases VM utilization but also limits the message transfer latency between the PEs by *collocating* neighboring PEs in the dataflow within the same VM. We order the PEs using a forward DFS traversal, rooted at the input PEs, and allocate resources to them in that order so as to increase the probability of collocating neighboring PEs. It should be noted that the CPU cores required for the individual PEs are not known in advance as the resource requirements depend on the current load which in turn depends on the resource requirements of the preceding PE. Hence, after assigning at least one CPU core to each PE (INCREMENTALLOCATION), the deployment algorithm chooses PEs in the order of largest bottlenecks in the dataflow, i.e., lowest relative PE throughput (Ω_i). This ensures that PEs needing more resources are chosen first for allocation. This in turn may increase the input rate (and processing load) on the successive PEs, making them the bottlenecks. As a result, we end up with an iterative approach to incrementally allocate CPU cores to PEs using the *VBP* heuristic until the throughput constraint is met. Since the resource allocation only impacts downstream PEs, this algorithm is bound to converge. We leave a theoretical proof to future work.

TABLE 1: Functions used in Initial Deployment Strategies

Function	Sharded (SH)	Centralized (CE)
GETCOSTOF-ALTERNATE	$A.cost$	$A.cost + S_i \times \sum successor.cost$
GETNEXTPE	**if** All PEs assigned **then** return $\underset{P_j \in \mathcal{P}}{\operatorname{argmin}}(\Omega_t^j)$ **else** **return** Next PE in DFS **end if**	
REPACKPEINSTANCES	N/A	Move PE instance to smallest VM big enough for required core-secs
REPACKFREEVMS	N/A	Iterative Repacking [30]

At the end, the algorithm performs two levels of repacking. After a solution is obtained using VMs from just the largest resource class, we first move one instance for all the over-provisioned PEs to the smallest resource class large enough to accommodate that PE instance (best fit, using REPACKPEINSTANCES). This may free up capacity on the VMs, and hence, we again use iterative repacking [30] (REPACKVMS) to repack all the VMs with spare capacity to minimize wasted cores. During this process, we may sacrifice instance collocation in favor of reduced resource cost. Our evaluation however shows that this is an acceptable trade-off toward maximizing the objective function. Note that these algorithms are all performed off-line, and the actual deployment is carried out only after these decision are finalized.

Both, the order in which PEs are chosen and the repacking strategy affects the quality of the heuristic. While the sharded strategy SH uses a local approach and does not perform any repacking, the centralized strategy CE repacks individual PEs and VMs, as shown in Table 1.

8.2 Runtime Adaptation Heuristic

The runtime adaptation kicks in periodically over the lifetime of the application execution. Alg. 2 considers the current state of the dataflow and cloud resources – available through monitoring – in adapting the alternate and resource selection. The monitoring gives a more accurate estimate of data rates, and hence the resource requirements and its cost.

As before, the algorithm is divided into two stages: Alternate selection and Resource allocation. However, unlike the deployment heuristic, we do not run both the stages at the same time interval. Instead, the alternates are selected every m intervals and the resources reallocated every n intervals. The former tries to switch alternates to achieve the throughput constraint given

the existing allocated resources, while the latter tries to balance the resources (i.e. provision new VMs or shutdown existing ones) given the alternates that are active at that time. Separating these stages serves two goals. First, it makes the algorithm for each stage more deterministic and faster since one of the parameters is fixed. Second, it reduces the number of retractions of deployment decisions occurring in the system. For e.g., if a decision to add a new VM leads to over-provisioning at a later time (but before the hourly boundary), instead of shutting down the VM, the alternate selection stage can potentially switch to an alternate with higher value, thus utilizing the extra resources available and in the process increase the application's value.

During the alternate selection stage, given the current data rate and resource performance, we first calculate the resources needed for each PE alternate (line 6). We then create a list of "feasible" alternates for a given PE, based on whether the current relative throughput is lesser or greater than the expected throughput $\widehat{\Omega}$. Finally, we sort the feasible alternates in decreasing order of the ratio between value to cost, and select the first alternate which can be accommodated using the existing resource allocation. After this phase the overall value either increases or decreases depending on whether the application was over-provisioned or under-provisioned to begin with, respectively.

The RESOURCEREDEPLOY procedure is used to allocate or de-allocate resources to maintain the required relative throughput while minimizing the overall cost. If the $\Omega \leq \widehat{\Omega} - \epsilon$, the algorithm proceeds similar to the initial deployment algorithm. It incrementally allocates additional resources to the bottlenecks observed in the system and repacks the VMs. However, if $\Omega > \widehat{\Omega} + \epsilon$, the system must scale in to avoid resource wastage and has two decisions to make: first, which PE needs to be scaled in and second, which instance of the PE is to be removed, thus freeing the CPU cores. The over-provisioned PE selected for scale in is the one with the maximum relative throughput (Ω). Once the over-provisioned PE is determined, to determine which instance of that PE should be terminated, we get the list of VMs on which these instances are running and then weigh these VMs using the following "weight" function (eqn 10). Finally, a PE instance which is executing on the least weighted VM is selected for removal.

$$\begin{aligned} VM\ Weight(r_i) = \quad & T_c(r_i) \times \left(\frac{FreeCores(r_i)}{TotalCores(r_i)}\right) \\ & \times (1 - \frac{\varphi_{r_i}}{\varphi}) \times \left(\frac{TotalCores(r_i) \times \pi}{Cost\ Per\ VM\ Hour}\right) \end{aligned} \quad (10)$$

where T_c is the time remaining in the current cost cycle (i.e. time till the next hourly boundary), φ_{r_i} is the number of PE instances for the over-provisioned PE on the VM r_i, and φ is the total number of instances for that PE across all VMs. The VM Weight is lower for VMs with less time left in their hourly cycle, and thus preferred for removal. This increases temporal utilization. Similarly, VMs with fewer cores used are prioritized for removal. Further, VMs with higher cost per normalized core have a lower weight so that they are selected first for shutdown. Hence the VM Weight metric helps us pick the PE instances in a manner that can help free up costlier, under-utilized VMs that can be shutdown at the end of their cost cycle to effectively reduce the resource cost.

Algorithm 2 Runtime Adaptation Heuristic Algorithm

```
1: procedure AlternateReDeploy(DataflowD, Ω_t)   ▷ Ω_t is the observed relative throughput
2:     for PE P ∈ D do                            ▷ Alternate selection phase
3:         alloc ← CurrAllocatedRes(P)            ▷ Gets the current allocated resources (accounting for Infra. variability)
4:         requiredC ← RequiredRes(P.activeAlternate)   ▷ Gets the required resources for the selected alternate
5:         for AlternateA ∈ P do
6:             requiredA ← ActualResRequirements(A)
7:             if Ω_t ≤ Ω̂ − ε then
8:                 if requiredA ≤ requiredC then   ▷ Select alternate with lower requirements
9:                     feasible.ADD(A)
10:                end if
11:            else if Ω_t ≥ Ω̂ + ε then
12:                if requiredA ≥ requiredC then   ▷ Select alternate with higher requirements
13:                    feasible.ADD(A)
14:                end if
15:            end if
16:        end for
17:        SORT(feasible)                          ▷ Decreasing order of value/cost
18:        for feasible alternate A do
19:            if requiredA < alloc then
20:                SwitchAlternate(A)
21:                done
22:            end if
23:        end for
24:    end for
25: end procedure
1: procedure ResourceReDeploy(DataflowD, Ω_t)
2:     if Ω_t ≤ Ω̂ − ε then
3:         Same procedure as initial deployment
4:     else if Ω_t ≥ Ω̂ + ε then
5:         while Ω ≥ Ω̂ do
6:             PE ← overProvisionedPE
7:             instance ← SelectInstanceToKill(PE)
8:             newAlloc ← RemovePEInstance(instance)
9:             Ω ← GetEstimatedThruput(D,newAlloc)
10:        end while
11:        repackFreeVMs()   ▷ Repack PEs in the VMs with free cores onto smaller VMs with collocation
12:    end if
13: end procedure
```

9 EVALUATION

We evaluate the proposed heuristics through a simulation study. To emulate real-world cloud characteristics, we extend CloudSim [31] simulator to *IaaSSim*, that incorporates temporal and spatial performance variability using VM and network performance traces collected from IaaS Cloud VMs [2]. Further, *FloeSim* simulates the execution of the dynamic dataflows [21], on top of IaaSSim, with support for dataflows, alternates, distributed deployment, runtime scaling, and plugins for different schedulers. To simulate data rate variations, the given data rate is considered as an average value and the instantaneous data rate is obtained using a random walk between $\pm 50\%$ of that value. However, to enable comparisons between different simulation runs, we generate this data trace once and use the same across all simulation runs.

2. IaaSSim and performance traces are available at http://github.com/usc-cloud/IaaSSimulator

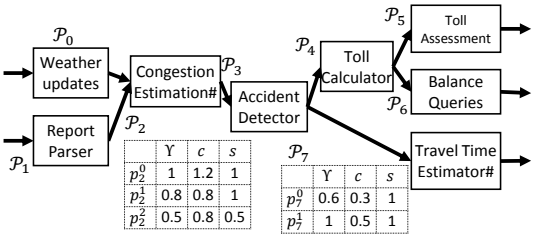


Fig. 5: Dynamic Continuous Dataflow for LRB. Alternates for $\mathcal{P}_2$ and $\mathcal{P}_7$ have value (γ), cost (c) and selectivity (s).

For each experiment, we deploy the Linear Road Benchmark (LRB) [16] as a dynamic dataflow using FloeSim, run it for 12 simulated hours ($T = 12\ hrs$) on simulated VMs whose performance traces were obtained from real Amazon EC2 VMs. Each experiment is repeated at least three times and average values are reported. We use a timestep duration of $t_s = 5\ mins$ at the start of which adaptation decisions are made.

9.1 Linear Road Benchmark (LRB)

The Linear Road Benchmark [16] is used to compare the performance of data stream management systems and has been adopted to general purpose stream processing systems [32]. Using this as a base, we develop a dynamic continuous dataflow application to evaluate the proposed scheduling heuristics. LRB models a road toll network within a confined area (e.g., 100 sq. miles), in which the toll depends on several factors including time of the day, current traffic congestion levels and proximity to accidents. It continuously ingests "position reports" from different vehicles on the road and is responsible for (i) detecting average speed and traffic congestion for a section, (ii) detecting accidents, (iii) providing toll notifications to vehicles whenever they enter a new section, (iv) answering account balance queries and toll assessments, and (v) estimating travel times between two sections. The goal is to support the highest number of expressways while satisfying the desired latency constraints. To simulate realistic road conditions, the data rate varies from around 10 msgs/sec to around 2,000 msgs/sec per expressway.

Fig. 5 shows the LRB benchmark implemented as a dynamic continuous dataflow. The *Weather Updates* ($\mathcal{P}_0$) and *Report Parse* ($\mathcal{P}_1$) PEs act as the input PEs for the dataflow. While the former receives low frequency weather updates, the latter receives extremely high frequency position reports from individual vehicles (each car sends a position report every 30 secs) and exhibits variable data rates based on the current traffic conditions. The *Congestion Estimation* PE ($\mathcal{P}_2$) estimates current as well as near-future traffic conditions for different sections of all the monitored expressways. This PE may have several alternates using different machine learning techniques that predict traffic with different accuracy and future horizons. We simulate three alternates with different value (γ), cost (c) and selectivity (s) values as shown in the tables in Fig. 5. The *Accident Detector* PE ($\mathcal{P}_3$) detects accidents based on the position reports, which is forwarded to *Toll Calculator* ($\mathcal{P}_4$) and *Travel Time Estimator* ($\mathcal{P}_7$) PEs. The former notifies toll values ($\mathcal{P}_5$) and account balances ($\mathcal{P}_6$) to the vehicles periodically. The latter ($\mathcal{P}_7$) provides travel time estimates, and has several alternates based on different forecasting models. For simulations, we use two alternates, e.g., (1) decision/regression tree model which takes several historical factors into account, and (2) time series models which predict using only the recent past traffic conditions.

9.2 Results

We compare the proposed centralized and sharded heuristics (CE and SH) and the GA algorithm with a brute force approach (BR) that explores the search tree but uses intelligent pruning to avoid searching sub-optimal or redundant sub-trees. We evaluate their overall profit achieved, overall relative throughput and the monetary cost of execution over the optimization period of $T = 12\ hrs$. An algorithm is better than another if it meets the necessary relative application throughput constraint, $\Omega \geq \widehat{\Omega} - \epsilon$, and has a higher value for the objective function Θ (Eqn. 8) Note that the necessary constraint for Ω *must* be met but higher values beyond the constraint do not indicate a better algorithm. Similarly, an algorithm with a higher Θ value is not better *unless* it also meets the Ω constraint.

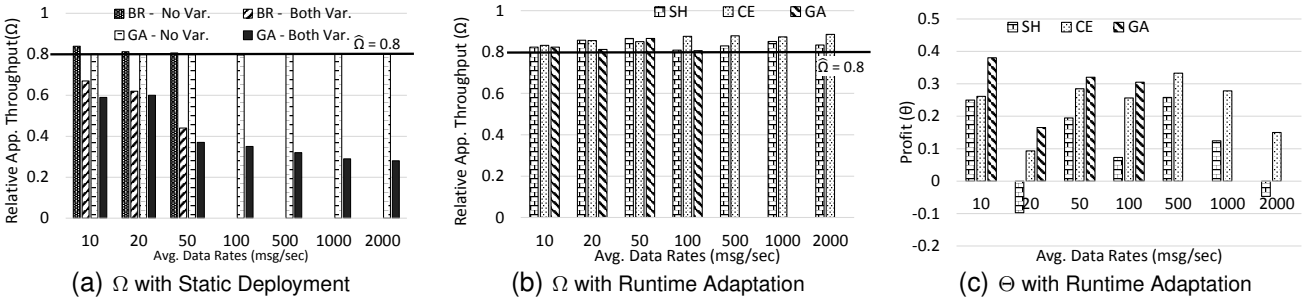
For all the experiments, we define the relative throughput threshold as $\widehat{\Omega} = 0.8$ with a tolerance of $\epsilon = 0.05$. We calculate σ for the LRB dataflow using Eqn. 9 by setting $C_{max}^{\Gamma_{min}} = \frac{0.5 \times T \times DataRate}{10}$ and $C_{max}^{\Gamma_{max}} = \frac{1.0 \times T \times DataRate}{10}$. We empirically arrive at these bounds by observing the actual break-even cost for executing the workflow using a brute force static deployment model.

1) Effect of Variability: Table 2 shows the overall profit and the relative throughput for a static deployment of LRB using different scheduling algorithms with an average input data rate of 50 msgs/sec. The overall profit which is a function of application value and resource cost remains constant due to a static deployment (without runtime adaptation). However, the relative throughput varies as we introduce infrastructure and data variability. In the absence of any variability, the brute force (BR) approach gives the best overall profit and also meets the throughput constraint ($\Omega \geq 0.8$). Further, GA approaches a near optimal solution with $\Theta_{GA} \rightarrow \Theta_{BR}$ when neither infrastructure nor input data rates vary, and is within the tolerance limit ($\widehat{\Omega} - \epsilon < \Omega = 0.79 < \widehat{\Omega} + \epsilon$). The SH and CE heuristics meet the throughput constraint but give a lower profit when there is no variability.

However, when running simulations with infrastructure and/or data variability, none of the approaches

TABLE 2: Effect of variability of infrastructure performance and input data rate on relative output throughput using different scheduling algorithms. Static LRB deployment with average of 50 msgs/sec input rate, $\widehat{\Omega} = 0.8$.

		Relative Application Throughput (Ω)			
Algo.	**Profit** (Θ)	Neither	Infra. Perf.	Data Rate	Both
BR	0.67	0.80	0.68	0.59	0.44
GA	0.65	0.79	0.67	0.48	0.37
SH	0.45	0.81	0.60	0.40	0.29
CE	0.58	0.81	0.66	0.42	0.31



(a) Ω with Static Deployment (b) Ω with Runtime Adaptation (c) Θ with Runtime Adaptation

Fig. 6: Effect of infrastructure and data rate variability on Static Deployment and Runtime Adaptation, as input data rate rises.

meet the desired throughput constraints with Ω values ranging between 0.29 to 0.68, which is much less than the goal of $\widehat{\Omega} = 0.8$. Since these experiments do not change the deployment at runtime, the initial deployment is based on assuming constant data rates and infrastructure performance. Even the two best static deployment strategies in the absence of variability, BR and GA, rapidly degrade in output throughput when both infrastructure and data variability are present. This is more so as the average input data rate increases from 10 msgs/sec to 2,000 msgs/sec in Fig. 6(a). Due to explosion in state space, we could not run the BR algorithm beyond 50 msgs/sec input rate. This analysis motivates the need for autonomic adaptations to application deployment.

2) Improvements with Runtime Adaptation: Figs. 6(b) and 6(c) show the improvements in relative throughput and overall application profit by utilizing different runtime adaptation techniques in the presence of both infrastructure as well as data variability. We use GA as the upper bound (instead of BR) since BR is prohibitively slow for runtime adaptations and, as shown in Table 2, GA approaches the optimal in many scenarios. However, as discussed later (Fig. 7(a)), even GA becomes prohibitive for data rates ≥ 500 msg/sec, and hence the missing entries in Figs. 6(b) and 6(c).

We observe that with dynamic adaptation both GA and the greedy heuristics (SH and CE) achieve the desired throughput constraint ($\Omega \geq \widehat{\Omega} = 0.8$) for all the input data rates tested. This allows us to compare their achieved application profit (Fig. 6(c)) and make several key observations. First, profit from GA is consistently more than the SH and CE greedy heuristics. Second, CE achieves a better profit than SH and reaches between 60% to 80% of GA's profit. In fact, SH gives negative profit (loss) in some cases. Understandably, the CE scheduler having a global view performs significantly better than SH that performs local optimizations. However, CE has a higher overhead due to centralized collection of monitoring data (the exact overhead is not available from simulations). Lastly, comparing the static and dynamic deployment from Table 2 and Fig. 6(c), for a data rate of 50 msg/sec under variable conditions, we do see a drop in profit using runtime adaptation as it tries to achieve the Ω constraint. For GA and CE, the profits for adaptive deployment drop from $0.65 \rightarrow 0.34$ and $0.58 \rightarrow 0.29$, respectively, but are still well above the break-even point of 0. But the static deployments violate the throughput constraint by a large margin which makes their higher profits meaningless.

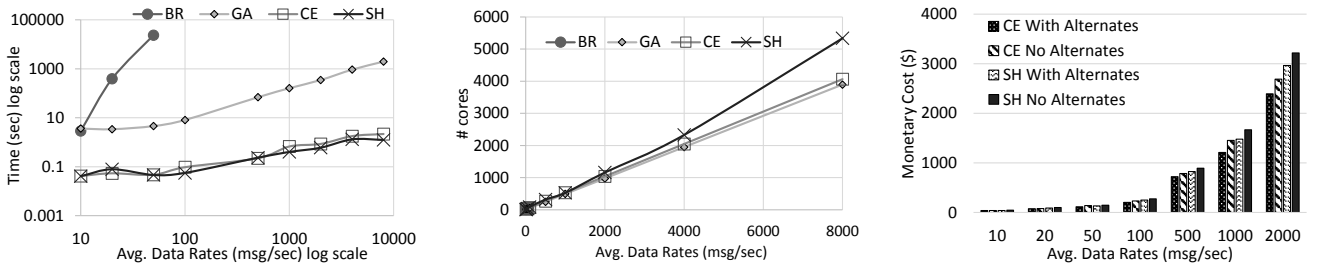
3) Scalability of Algorithms: Figs. 7(a) and 7(b) show algorithm scalability with respect to algorithm runtime and the number of cores for the initial deployment algorithm with increase in the incoming data rates. While the BR and the GA algorithms provide (near) optimal solutions for smaller data rates, their overhead is prohibitively large for higher data rates (fig. 7(a)) ($> 10,000$ secs for BR at 50 msg/sec, and $> 1,000$ secs for GA at 8,000 msg/sec). This is due to the search space explosion with the increase in the number of required VMs as shown in fig 7(b). On the other hand, both CE and SH greedy heuristics take just ~ 2.5 secs to compute at 8,000 msg/sec, and scale linearly ($O(|\varphi| + |R|)$) with the number of PE instances ($|\varphi|$) and number of virtual machines($|R|$). Further we see that SH algorithm leads to higher resource wastage (more cores) with increase in the data rates, while CE and GA show a linear relation to the data rates in Fig. 7(b). Similar results are seen for the adaptation stage for SH and CE algorithms but the plots are omitted due to space constraints.

4) Benefit of using Alternates: We study the reduction in monetary cost to run the continuous dataflow due to the use of alternates, as opposed to a dataflow deployed with only a single implementation for the PEs; we choose the implementation with the highest value ($\Gamma = 1$). Fig. 7(c) shows the cost (US$) of resources required to execute the LRB dataflow for the optimization interval $T = 12\ hr$ using the greedy heuristics with runtime adaptation, in the presence of both infrastructure and data variability. We use AWS's EC2 prices using m1.* generation of VMsfor calculating the monetary cost.

We see that the use of alternates by runtime adaptation leads to a reduction in total monetary cost by 6.9% to 27.5%, relative to the non-alternate dataflow; alternates with different processing requirements provide an extra dimension of control to the scheduler. In addition, the benefit of alternates increases with high data rate – as the input data rate and hence the resource requirement increases, even small fluctuations in data rate or infrastructure performance causes new VMs to be acquired to meet the Ω constraint, and acquiring VMs has a higher overhead (e.g., the hourly cost cycle and startup overheads) than switching between alternates.

10 Related Work

Scientific workflows [33], continuous dataflow systems [7], [8], [5], [9] and similar large-scale distributed



(a) Algorithm time as a function of datarate. (b) # Cores as a function of datarate (c) Monetary cost benefit of using alternates

Fig. 7: Algorithm scalability (a,b) and advantage of alternates (c)

programming frameworks [34], [35] have garnered a renewed research focus due to the recent explosion in the amount of data, both archived and real time, and the need for large-scale data analysis on this "Big Data". Our work is based upon the stream processing and continuous dataflow systems that allow a task-graph based programming model to execute long running continuous applications which process incoming data streams in near-real time. Other related work includes flexible workflows and heterogeneous computing, service oriented architecture (SOA) and autonomic provisioning and resource management in clouds. We discuss the state-of-the-art in each of these research areas.

Continuous dataflow systems have their root in Data Stream Management Systems, which process continuous queries over tuple streams composed of of well-defined operators [36]. This allows operator-specific query optimizations such as operator split and merge to be performed [37]. General-purpose continuous dataflow systems such as S4 [7], Storm [8], and Spark [34], on the other hand, allow user-defined processing elements, making it necessary to find generic auto-scaling solutions such as operator scaling and data-parallel operations [38]. Several solutions, including one leveraging cloud elasticity to support auto-scaling, have been proposed [11]. However, these systems [10], [39], only consider data variability as a factor for auto-scaling decisions and assume that the underlying infrastructure offers the same performance over time. Our work shows that this assumption does not hold in virtualized clouds.

Autonomic provisioning for workload resource management on clouds have been proposed. These use performance monitoring and model-based approach [40], [41]. We use a similar approach and propose heuristics for dynamic continuous dataflows that handle not only data rate variations but also changes in the underlying infrastructure performance. Recent work [32] integrates elastic scale out and fault tolerance for stateful stream processing but adopts a local only policy based on CPU utilization for scaling. In this article, we assume stateless PEs, and fault tolerance is beyond the scope of this work. Our results do show that using local scale-out strategies that ignore the dataflow structure under-perform, and hence motivates heuristics with a global view.

Flexible workflows [42], [19] and service selection in SOA [43] allow workflow compositions to be transformed at runtime. This provides a powerful compositional tool to the developer to define business-rule based generic workflows that can be specialized at runtime depending on the environmental characteristics. The notion of "alternates" we propose is similar in that it offers flexibility to the developer and a choice of execution implementations at runtime. However, unlike flexible workflows where the decision about task specialization is made exactly once based on certain deterministic parameters, in continuous dataflows, this decision has to be re-evaluated regularly due to their continuous execution model and dynamic nature of the data streams.

To exploit a heterogeneous computing environment, an application task may be composed of several subtasks that have different requirements and performance characteristics. Various dynamic and static task matching and scheduling techniques have been proposed for such scenarios [18], [44], [45]. The concept of alternates in dynamic dataflow is similar to these. However, currently, we do not allow heterogeneous computing requirements for these alternates, though they may vary in processing requirements. Even with this restriction, the concept of alternates provides a powerful programming abstraction that allows us to switch between them at runtime to maximize the overall utility of the system in response to changing data rates or infrastructure performance.

Several studies have compared the performance of the virtualized environment against the barebones hardware to show their average performances are within an acceptable tolerance limit of each other. However these studies focused on the average performance characteristics and not on the variations in performance. Recent analysis of public cloud infrastructure [12], [46], [47], [14], [48] demonstrate high fluctuations in various cloud services, including cloud storage, VM startup and shutdown time as well as virtual machines core performance and virtual networking. However, the degree of performance fluctuations vary across private and different public cloud providers [13]. Reasons for this include multi-tenancy of VMs on the same physical host, use of commodity hardware, collocation of faults, and roll out of software patches to the cloud fabric. Our own studies confirm these. On this basis, we develop an abstraction of the IaaS cloud that incorporates infrastructure variability and also include it in our IaaS Simulator.

Meta-heuristics have been widely used to address the task scheduling problem [24], [25], [26], [27]. Most of the approaches are nature inspired and rely on GA [28], ant colony optimization [49], particle swarm optimization [50] or simulated annealing [51] techniques to search

for sub-optimal solutions. A number of studies to analyze the efficiency of meta-heuristics [24] show that in certain scenarios GAs can over perform greedy heuristics. More generally, Zamfirache et. al. [27] show that population based GA meta-heuristics of classical greedy approaches provide, through mutations, solutions that are better compared to their classic versions. Recently, a comparison of ant colony optimization and particle swarm optimization with a GA for scheduling DAGs on clouds was proposed [26]. None of these task scheduling algorithms or meta-heuristics based solutions take into account a dynamic list of task instances. Our own prior work [25] uses a GA to elastically adapt the number of task instances in a workflow to incoming web traffic but does not consider alternates or performance variability.

11 Conclusion

In this article, we have motivated the need for online monitoring and adaptation of continuous dataflow applications to meet their QoS constraints in the presence of data and infrastructure variability. To this end we introduce the notion of **dynamic dataflows**, with support for alternate implementations for dataflow tasks. This not only gives users the flexibility in terms of application composition, but also provides an additional dimension of control for the scheduler to meet the application constraints while maximizing its value.

Our experimental results show that the continuous adaptation heuristics which makes use of application dynamism can reduce the execution cost by up to 27.5% on clouds while also meeting the QoS constraints. We have also studied the feasibility of GA based approach for optimizing execution of dynamic dataflows and show that although the GA based approach gives near-optimal solutions its time complexity is proportional to the input data rate, making it unsuitable for high velocity applications. A hybrid approach which uses GA for initial deployment and the CE greedy heuristic for runtime adaptation may be more suitable. This is to be investigated as future work.

In addition, we plan to extend the concept of dynamic tasks which will further allow for alternate implementations at coarser granularity such as "alternate paths", and provide end users with more sophisticated controls. Further, we plan to extend the resource mapping heuristics for an ensemble of dataflows with a shared budget and address issues such as fairness in addition to throughput constraint and application value.

Acknowledgments

This material is based on research sponsored by DARPA, and the Air Force Research Laboratory under agreement number FA8750-12-2-0319. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, and the Air Force Research Laboratory or the U.S. Government.

References

[1] L. Douglas, "3d data management: Controlling data volume, velocity, and variety," Gartner, Tech. Rep., 2001.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Communications*. ACM, 2008, vol. 51, no. 1, pp. 107–113.

[3] M. Gatti, P. Cavalin, S. B. Neto, C. Pinhanez, C. dos Santos, D. Gribel, and A. P. Appel, "Large-scale multi-agent-based modeling and simulation of microblogging-based online social network," in *Multi-Agent-Based Simulation XIV*. Springer, 2014, pp. 17–33.

[4] I. Fronza, A. Sillitti, G. Succi, M. Terho, and J. Vlasenko, "Failure prediction based on log files using random indexing and support vector machines," in *Journal of Systems and Software*. Elsevier, 2013, vol. 86, no. 1, pp. 2–11.

[5] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "Ibm infosphere streams for scalable, real-time, intelligent transportation services," in *International Conference on Management of data*. ACM SIGMOD, 2010, pp. 1093–1104.

[6] Y. Simmhan, S. Aman, A. Kumbhare, R. Liu, S. Stevens, Q. Zhou, and V. Prasanna, "Cloud-based software platform for big data analytics in smart grids," in *Computing in Science Engineering*, July 2013, vol. 15, no. 4, pp. 38–47.

[7] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *IEEE International Conference on Data Mining Workshops (ICDMW)*, 2010.

[8] "Storm: Distributed and fault-tolerant realtime computation," http://storm.incubator.apache.org/, accessed: 2014-06-30.

[9] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *USENIX conference on Hot Topics in Cloud Ccomputing*, 2012, pp. 10–10.

[10] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Esc: Towards an elastic stream computing platform for the cloud," in *IEEE International Conference on Cloud Computing (CLOUD)*, July 2011, pp. 348–355.

[11] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," in *Transactions on Parallel and Distributed Systems*. IEEE, 2012, vol. 23, no. 12, pp. 2351–2365.

[12] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2011.

[13] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: comparing public cloud providers," in *conference on Internet measurement*. ACM SIGCOMM, 2010, pp. 1–14.

[14] I. Moreno, P. Garraghan, P. Townend, and J. Xu, "Analysis, modeling and simulation of workload patterns in a large-scale utility cloud," in *Transactions on Cloud Computing*. IEEE, April 2014, vol. 2, no. 2, pp. 208–221.

[15] A. Kumbhare, Y. Simmhan, and V. K. Prasanna, "Exploiting application dynamism and cloud elasticity for continuous dataflows," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 57.

[16] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *international conference on Very Large Databases*. VLDB Endowment, 2004, pp. 480–491.

[17] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," in *Distributed and parallel databases*. Springer, 2003, vol. 14, no. 1, pp. 5–51.

[18] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," in *Journal of Parallel and distributed Computing*. Elsevier, 1999, vol. 59, no. 2, pp. 107–131.

[19] J. Wainer and F. de Lima Bezerra, "Constraint-based flexible workflows," in *Groupware: Design, Implementation, and Use*. Springer, 2003, pp. 151–158.

[20] C. Ouyang, E. Verbeek, W. M. Van Der Aalst, S. Breutel, M. Dumas, and A. H. Ter Hofstede, "Formal semantics and analysis of control flow in ws-bpel," in *Science of Computer Programming*. Elsevier, 2007, vol. 67, no. 2, pp. 162–198.

[21] Y. Simmhan and A. G. Kumbhare, "Floe: A continuous dataflow framework for dynamic cloud applications," *CoRR*, vol. abs/1406.5977, 2014.

[22] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl, "Qos aggregation for web service composition using workflow patterns," in *IEEE*

International conference on Enterprise distributed object computing., 2004, pp. 149–159.
[23] G. J. Woeginger, "Exact algorithms for np-hard problems: A survey," in *Combinatorial Optimization–Eureka, You Shrink!* Springer, 2003, pp. 185–207.
[24] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," in *Journal of Parallel and Distributed Computing*. Elsevier, Jun. 2001, vol. 61, no. 6, pp. 810–837.
[25] M. E. Frincu, "Scheduling highly available applications on cloud environments," in *Future Generation Computer Systems*. Elsevier, 2014, vol. 32, pp. 138–153.
[26] Z. Wu, X. Liu, Z. Ni, D. Yuan, and Y. Yang, "A market-oriented hierarchical scheduling strategy in cloud workflow systems," in *The Journal of Supercomputing*. Springer, 2013, vol. 63, no. 1, pp. 256–293.
[27] F. Zamfirache, M. Frincu, and D. Zaharie, "Population-based metaheuristics for tasks scheduling in heterogeneous distributed systems," in *Numerical Methods and Applications*. Springer, 2011, vol. 6046, pp. 321–328.
[28] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Cambridge, MA, USA: MIT Press, 1992.
[29] D. E. Goldberg and K. Deb, "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms," in *Foundations of Genetic Algorithms*, 1990, pp. 69–93.
[30] J. Kang and S. Park, "Algorithms for the variable sized bin packing problem," in *European Journal of Operational Research*. Elsevier, 2003, vol. 147, no. 2, pp. 365–372.
[31] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," in *Software: Practice and Experience*. Wiley Online Library, 2011, vol. 41, no. 1, pp. 23–50.
[32] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *International conference on Management of data*. ACM SIGMOD, 2013, pp. 725–736.
[33] J. Yu and R. Buyya, "A taxonomy of scientific workflow systems for grid computing," in *Sigmod Record*. ACM, 2005, vol. 34, no. 3, p. 44.
[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *USENIX conference on Hot topics in cloud computing*, 2010.
[35] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce," in *IEEE International Conference on Cluster Computing*. IEEE, 2009.
[36] S. Babu and J. Widom, "Continuous queries over data streams," in *Sigmod Record*. ACM, 2001, vol. 30, no. 3, pp. 109–120.
[37] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *International Conference on Data Engineering*. IEEE, 2003, pp. 25–36.
[38] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu, "Efficient dynamic operator placement in a locally distributed continuous query system," in *On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*. Springer, 2006, pp. 54–71.
[39] R. Tolosana-Calasanz, J. Angel Bañares, C. Pham, and O. Rana, "End-to-end qos on shared clouds for highly dynamic, large-scale sensing data streams," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2012, pp. 904–911.
[40] A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma, "Towards autonomic workload provisioning for enterprise grids and clouds," in *IEEE/ACM International Conference on Grid Computing*, Oct 2009, pp. 50–57.
[41] Y. Hu, J. Wong, G. Iszlai, and M. Litoiu, "Resource provisioning for cloud computing," in *Conference of the Center for Advanced Studies on Collaborative Research*. Riverton, NJ, USA: IBM Corp., 2009, pp. 101–111.
[42] G. J. Nutt, "The evolution towards flexible workflow systems," in *Distributed Systems Engineering*. IOP Publishing, 1996, vol. 3, no. 4.
[43] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services," in *Transactions on Services Computing*. IEEE, 2010, vol. 3, no. 3, pp. 223–235.
[44] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," in *Journal of Parallel and Distributed Computing*. Elsevier, 1997, vol. 47, no. 1, pp. 8–22.
[45] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," in *Transactions on Parallel and Distributed Systems*. IEEE, 2002, vol. 13, no. 3, pp. 260–274.
[46] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," in *Transactions on Parallel and Distributed Systems*. IEEE, 2011, vol. 22, no. 6, pp. 931–945.
[47] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance analysis of high performance computing applications on the amazon web services cloud," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2010, pp. 159–168.
[48] Z. Ou, H. Zhuang, A. Lukyanenko, J. Nurminen, P. Hui, V. Mazalov, and A. Yla-Jaaski, "Is the same instance type created equal? exploiting heterogeneity of public clouds," in *Transactions on Cloud Computing*. IEEE, July 2013, vol. 1, no. 2, pp. 201–214.
[49] A. Colorni, M. Dorigo, and V. Maniezzo, "Distributed Optimization by Ant Colonies," in *European Conference on Artificial Life*, 1991, pp. 134–142.
[50] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *IEEE International Conference on Neural Networks*, vol. 4, 1995, pp. 1942–1948 vol.4.
[51] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," in *The Journal of Chemical Physics*. AIP, 1953, vol. 21, no. 6, pp. 1087–1092.

Alok Gautam Kumbhare is a PhD candidate in Computer Science at the Univ. of Southern California and a Research Assistant in the DARPA XDATA project. His research interests are in the areas of resource management and fault tolerance for workflow and dataflows on distributed environment such as Clouds.

Yogesh Simmhan is an Assistant Professor at the Indian Institute of Science. Previously, he was a Research Assistant Professor in Electrical Engineering at the University of Southern California and a postdoc at Microsoft Research. His research explores scalable abstractions, algorithms and applications for distributed platforms, and helps advance fundamental knowledge while offering a practitioner's insight. He has a Ph.D. in Computer Science from Indiana University. Senior member of ACM and IEEE.

Marc Frincu received his Ph.D. from West University of Timisoara Romania in 2011. During his Ph.D. he also worked as a junior researcher at the e-Austria Research Institute Romania. In 2012 he joined University of Strasbourg France as a postdoctoral researcher focusing on cloud scheduling. In 2013 he took a postdoctoral research associate position at Univ of Southern California dealing with cloud computing and smart grid systems.

Viktor K. Prasanna is the Powell Chair in Engineering, Professor of Electrical Engineering and Computer Science, and Director of the Center for Energy Informatics at the Univ of Southern California. His research interests include HPC, Reconfigurable Computing, and Embedded Systems. He received his MS from the School of Automation, Indian Institute of Science and PhD in Computer Science from Penn State. He is a Fellow of IEEE, ACM and AAAS.

Exploiting Application Dynamism and Cloud Elasticity for Continuous Dataflows

Alok Kumbhare, Yogesh Simmhan and Viktor K. Prasanna
University of Southern California
Los Angeles, California 90089
{kumbhare, simmhan, prasanna}@usc.edu

ABSTRACT

Contemporary continuous dataflow systems use elastic scaling on distributed cloud resources to handle variable data rates and to meet applications' needs while attempting to maximize resource utilization. However, virtualized clouds present an added challenge due to the variability in resource performance – over time and space – thereby impacting the application's QoS. Elastic use of cloud resources and their allocation to continuous dataflow tasks need to adapt to such infrastructure dynamism. In this paper, we develop the concept of "dynamic dataflows" as an extension to continuous dataflows that utilizes *alternate tasks* and allows additional control over the dataflow's cost and QoS. We formalize an optimization problem to perform both deployment and runtime cloud resource management for such dataflows, and define an objective function that allows trade-off between the application's *value* against resource *cost*. We present two novel heuristics, local and global, based on the variable sized bin packing heuristics to solve this NP-hard problem. We evaluate the heuristics against a static allocation policy for a dataflow with different data rate profiles that is simulated using VM performance traces from a private cloud data center. The results show that the heuristics are effective in intelligently utilizing cloud elasticity to mitigate the effect of both input data rate and cloud resource performance variabilities on QoS.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.1.3 [**Concurrent Programming**]: Distributed programming; I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods

General Terms

Design, Performance, Algorithms

SC'13, November 17-21 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2503210.2503240

Keywords

Dataflows, clouds, resource management, optimization, data velocity, runtime adaptation

1. INTRODUCTION

Distributed platforms offer a vital solution space to address problems on "Big Data" [9]. Data analytics platforms have expanded beyond scaling for *volume* and *variety* dimensions of data and started focusing on continuous data with high and uneven data rates – the *velocity* dimension. Stream processing frameworks such as Yahoo's S4 [26], Twitter's Storm, IBM InfoSphere Streams [5], and Spark Streaming [47] provide continuous dataflow programming abstractions to build and deploy tasks graphs as distributed applications at scale on commodity clusters and clouds.

While these systems support high input data rates, they fail to adequately consider fluctuations in the data rates that are observed in the real world. Some statically over- or under-provision resources depending on availability, and consequently trade-off between cost of unused resources during low data rate period and the penalty of high processing latencies during the high data rate period. Others track the changes to the incoming data rates and redeploy the application, but this is not done online and can cause loss of messages or high processing latencies during the switch. Cloud computing platforms offer the ability to elastically acquire and release resources, and some recent continuous dataflow frameworks like Esc [32], WSAggregation [14] and others [17, 12] leverage these to perform online scaling of resources based on application load. Few of them also support active migration of processing elements between resources to improve performance and to maximize resource utilization.

Despite these advances, there are two clear gaps in current research on executing continuous dataflows on elastic public clouds: (1) adapting to changing performance of cloud resources, and (2) offering more flexible cost-benefit trade-offs to users running such dataflows on (commercial) clouds.

Cloud infrastructure exhibits performance variability over time and space. The same virtual machine (VM) instance has different resource characteristics over time due to multitenancy and changing workloads in the data center. Different VM instances of the same resource class (e.g. Medium) show different performance due to placement and diversity in commodity hardware [16]. As a result, scheduling strategies that assume deterministic and homogeneous cloud behavior fail to deliver expected Quality of Service (QoS) to the dataflow applications running on clouds.

In this paper, we address these issues by introducing a

novel concept of "dynamic dataflows", where each task in the dataflow consists of one or more alternate implementations with different cost-benefit ratio. At a given time, the system is free to choose any one of these implementations based on the domain policies and execution metrics. This concept of alternate tasks in dataflows provide another dimension of control, in addition to resource elasticity, which can be leveraged to achieve desired execution goals and constraints in distributed streaming applications.

Utilizing the dynamic dataflow paradigm, we further propose a multi-objective optimization problem to balance the execution cost, processing latency and "value" for the dataflow application. We develop a set of deployment and runtime heuristics that tune both the elastic scaling and the selection of alternate tasks to adapt to the fluctuations in the data rate, as well as, to mitigate the effect of performance variability in the cloud infrastructure.

We evaluate the heuristics through simulations that use real performance traces obtained from a medium-sized private cloud deployment and the empirical results show that the cost of execution for a dataflow can be reduced by as much as 15% given various tolerances in the application throughput and application value.

The rest of the paper is organized as follows. We survey and compare related work in section 2. Section 3 describes the continuous dataflow application model with extensions for dynamic dataflows. Further, we discuss the infrastructure model and parameters in Section 4, with focus on commodity clusters and public clouds. Section 5 presents our approach towards dataflow deployment and runtime adaptation using cloud elasticity and alternate tasks. We formalize the optimization problem in Section 6 followed by multivariate deployment and runtime optimization heuristics in Section 7. We offer implementation details of the dataflow execution and cloud infrastructure simulation in Section 8.1 which is evaluated for various scenarios, as presented in Section 8.2. Section 9 concludes the paper with outlook on future work.

2. RELATED WORK

Scientific workflows [42, 4, 22, 35, 3], continuous dataflow systems [8, 1, 25, 28] and similar large scale distributed programming frameworks [46, 29] have garnered a renewed research focus due to the recent explosion in the amount of data, both archived and real time, and the need for large scale data analysis. Our work is closely related to the stream processing and continuous dataflow systems that allow a task-graph based programming model to execute long running continuous applications which process incoming data streams in near-real time. Other related work includes flexible workflows, service oriented architecture and the notion of "alternates" in heterogeneous computing paradigm. We discuss state-of-the-art in each of these research areas in relation to our proposed system.

2.1 Continuous dataflows, and existing auto scaling techniques

Continuous dataflow systems have their root in Data Stream Management Systems, the primary focus of which is to process continuous queries over tuple streams, composed using a set of welldefined operators [2]. This allows operator-specific query optimizations such as operator split and merge to be performed [33]. General purpose continuous dataflow systems such as S4 [26], Storm[1], and Spark[46], on the other hand, allow arbitrary processing elements, making it necessary to find generic auto scaling solutions such as operator scaling and data parallel operations [48]. Several solutions, including one leveraging cloud elasticity to support auto scaling, have been proposed [12]. However, most of these systems [32, 36], only consider data variability as a factor for auto scaling decisions and assume that the underlying infrastructure offers same performance overtime. Our work shows that this assumption does not hold in virtualized cloud systems and proposes heuristics that handle not only the variations in data rates but also changes in the underlying infrastructure performance.

2.2 Workflow and Task Scheduling

Task based scheduling have been extensively studied and various heuristics proposed. These algorithms are broadly classified into list-scheduling algorithms, clustering, and search based algorithms. Of these, list-scheduling algorithms have received the most attention. The goal of these algorithms is to produce a prioritized list of tasks and place each on the processing unit to minimize a certain cost function (or alternatively, maximize a certain utility function). These algorithms are generally defined for a bounded (and fixed) set of tasks and processors. Our global heuristic is similar to these algorithms; however, the list of task instances as well as their priorities are dynamic and based on external factors.

In addition, several specialized algorithms have been proposed for mapping workflows on to cloud resources with the goal of minimizing the makespan as well as the resource cost [44, 43, 30, 20]. Others use a mix of cloud and local cluster resources to balance performance and cost. However, none of these algorithms consider infrastructure variability of the cloud platform as a factor in scheduling.

2.3 Flexible workflows and SOA

Flexible workflows [27, 31, 39] and service selection in Service Oriented Architecture (SOA) [45, 24, 11] allow workflow compositions to be transformed at runtime. This provides a powerful compositional tool to the developer to define business rule based generic workflows that can be specialized at runtime based on the environmental characteristics. The notion of "alternates" we propose is similar to this in that it allows greater flexibility to the developer and a choice of execution at runtime. However, unlike flexible workflows where the decision about task specialization is made exactly once based on certain deterministic parameters, in continuous dataflows, this decision has to be re-evaluated regularly due to their continuous execution model and dynamic nature of the data streams.

2.4 Heterogeneous computing

To exploit a heterogeneous computing environment, an application task may be composed of several subtasks that have different requirements and performance characteristics. Various dynamic and static task matching and scheduling techniques have been proposed for such scenarios [23, 40, 37]. The concept of alternates in dynamic dataflow is similar to these; however, currently, we do not allow heterogeneous computing requirements for these alternates, though they

[1] `HTTP://storm-project.net/`

may vary in processing requirements. Even with this restriction, the concept of alternates provides a powerful programming abstraction that allows us to switch between them at runtime to maximize the overall utility of the system in response to changing data rates or infrastructure performance.

2.5 Cloud Infrastructure Performance analysis and modelling

Several studies have compared the performance of the virtualized environment against the barebones hardware to show their average performances are within an acceptable tolerance limit of each other. However these studies focused on the average performance characteristics and not on the variations in performance. Recent analysis of public cloud infrastructure [16, 15, 18] demonstrate high fluctuations in various cloud services, including cloud storage, VM startup and shutdown time as well as virtual machines core performance and virtual networking. Reasons for this include multi-tenancy of VMs on the same physical host, placement of VMs in the data center, use of commodity hardware, collocation of faults, and roll out of software patches to the cloud fabric, among others. Our own studies confirm these. On this basis, we develop an abstraction of the IaaS cloud that incorporates infrastructure variability.

3. DYNAMIC DATAFLOW APPLICATION MODEL

Directed acyclic task (DAG) graphs are familiar constructs used to model loosely as well as tightly coupled applications, and have been adopted by the scientific workflow and dataflow communities, among others. We extend this model to formally define dynamic dataflows that operate on continuous data streams and offer task alternatives.

DEF. 1 (CONTINUOUS DATAFLOW). *A continuous dataflow G is a quadruple $G = \{P, E, I, O\}$, where $P = \{P_1, P_2, ..., P_n\}$ is the set of* Processing Elements (PE) *and $E = \{(P_i, P_j) \mid P_i, P_j \in P\}$ is a set of dataflow edges such that there is no cycle, where (P_i, P_j) denotes flow of data* messages *from P_i to P_j. $I \neq \varnothing \subset P$ is a set of input PEs where external data messages enter the dataflow continuously with variable rates, and $O \neq \varnothing \subset P$ is a set of output PEs which emit data messages to be consumed by an external entity.*

Each PE represents a long-running user-defined task in the dataflow which executes continuously, accepting and consuming data messages from the incoming edges and producing messages on the outgoing edges. A directed edge between two PEs connects an output port from the source PE to an input port of the sink PE, and represents a flow of messages between the two [7]. A user can select different semantics for the flow of data on the edges, such as sequence, and-split, synchronize, merge, choice, multi-choice, and so on, similar to dataflow patterns in static workflows [38]. Without loss of generality, we make the simplifying assumption of *and-split semantics* for edges originating from the same output port of a PE (i.e., output messages on a port are duplicated on all outgoing edges) and multi-merge [38] semantics for edges terminating at an input port of another PE (i.e., input messages on a port from all incoming edges are interleaved).

We extend continuous dataflows to "Dynamic Dataflows" by incorporating the concept of alternative PEs (or simply, *alternates*). In heterogeneous computing [23] and flexible (single execution) workflows [39], alternates are *selected once* at runtime and thereafter their dataflow path of execution is fixed. We advance this concept to continuous dataflow where alternate selection is an ongoing process. Alternates enable the user to optionally define alternative implementations for a given PE, each of which may possess different performance characteristics, resource needs and quality of service and hence offer different relative trade-offs between cost and QoS. At the same time, the choice of alternates does not impact the correctness of the application. Due to space constraints, a detailed discussion of "alternates" and the programming model is beyond the scope of this paper. We assume that the alternates have been specified using an appropriate programming model and the set of feasible alternates is available to the proposed heuristics. Alternates defined for different PEs operate independently, and hence any combination of them may be selected. It is also valid to replace one alternate with another during runtime.

DEF. 2 (DYNAMIC DATAFLOWS). *A Dynamic Dataflow D is a continuous dataflow where each PE $P_i \in P$ has set of alternate implementations $P_i = \{p_i^1, p_i^2, ...p_i^j \mid j \geq 1\}$.*

We capture the QoS, cost and performance for each alternate of the PE using a set of metrics, viz., the relative value (γ_i^j), the alternate cost (c_i^j) in core-seconds per message, and the selectivity (s_i^j). The *relative value*, $0 < \gamma_i^j \leq 1$, for an alternate p_i^j is defined as $\gamma_i^j = \frac{f(p_i^j)}{\text{MAX}_j\{f(p_i^j)\}}$, where $f(p_i^j)$ is a user-defined value function for the alternate (e.g., the F_1 statistical measure for a classification PE). It quantifies the relative benefit for the user of picking this alternate. The *processing cost*, c_i^j, is the time (in seconds) required to process a single message on a "standard" CPU core (see § 4) for the alternate p_i^j . The *selectivity*, s_i^j, is the ratio of number of the output messages produced to the number of input messages consumed by the alternate p_i^j to complete a logical unit of operation. The selectivity of a PE helps determine the downstream data rate in the dataflow.

A dataflow may be initially deployed with a particular configuration of alternates which can then be switched during runtime to meet the application's constraints. To keep the problem tractable, we enforce that these changes are only made at periodic intervals. During a given time interval t, only one of the alternates for a PE P_i is active, given by:

$$A_i^j(t) = \begin{cases} 1 & \text{if } p_i^j \text{ is active during interval } t \text{ for PE } P_i \\ 0 & \text{otherwise} \end{cases}$$

and $\sum_{p_i^j \in P_i} A_i^j(t) = 1$

The value of the PE P_i in time interval t is then given by:

$$\Gamma_i(t) = \sum_{p_i^j \in P_i} \left(\gamma_i^j \cdot A_i^j(t)\right)$$

We obtain the value for the entire dataflow application during time interval t by aggregating values of the individual alternates since value can be perceived as an additive property [19] over the application graph.

DEF. 3 (NORMALIZED APPLICATION VALUE). *The normalized application value $0 < \Gamma(t) \leq 1$ for a dynamic dataflow*

D during time interval t is given by $\Gamma(t) = \frac{\sum_{P_i \in P} \Gamma_i(t)}{|P|}$, *where* $|P|$ *is the number of PEs in dynamic dataflow.*

The application's value thus obtained gives an indication of its overall quality from the user's perspective and can be considered as one dimension of QoS. Yet another dimension, particularly in the context of continuous dataflows, is the observed application throughput. However, raw application throughput is not meaningful because it is a function of the incoming data rates during that time interval. We use the *Relative application throughput*, Ω, as a metric of application performance.

As before, the relative application throughput is built up from the relative PE throughputs of the individual alternates, given by $0 < \Omega_i(t) \leq 1$ for a PE P_i during interval t. It is defined as the ratio of current output data rate (absolute throughput) $o_i(t)$ to the maximum achievable output data rate $o_i^{max}(t)$ given by $\Omega_i(t) = \frac{o_i(t)}{o_i^{max}(t)}$. The output data rate for a PE depends on its alternates' selectivity, and bound by its input data rate and the resources available to process the inputs. The maximum output throughput is achieved when there are enough resources for P_i to process all incoming data messages within the interval t, given by $o_i^{max} = i_i \cdot s_i^j$, where i_i is the input date rate, and s_i^j is the selectivity of its alternate (omitting the time intervals for brevity). The actual output data rate is thus given by $o_i = \frac{min\left(i_i \Delta t, \frac{\phi_i \Delta t}{c_i^j}\right) \cdot s_i^j}{\Delta t}$, where ϕ_i is its total CPU core allocation given by $\phi_i = \sum_j \pi_j$, (where π_j is the core coefficient of the allocated core and j $\in$ allocated cores for the PE (see § 4), and c_i^j is its latency to process a message on a standard core (with $\pi_j = 1$).

The relative throughput for the application is not additive as it depends on the critical processing path of the PEs. We treat the entire dataflow as a black box consisting of a single PE, and use the accumulative incoming data rate at the input PEs and the accumulative output rate at the output PEs to obtain the relative application throughput.

Def. 4 (Relative Application Throughput). *The relative application throughput* $0 < \Omega(t) \leq 1$ *for dataflow D during time t is given by* $\Omega(t) = \frac{\sum_{P_i \in O} \Omega_i(t)}{|O|}$ *where O is the set of output tasks.*

The dynamic dataflows thus extend the familiar DAG application model with support for continuous data streams and dynamic PE alternates. Application value and relative throughput provide complimentary metrics to assess overall application QoS, and allow for clear trade-offs to be made with the resource costs when optimizing the resource scaling and alternate selection.

4. CLOUD INFRASTRUCTURE MODEL

Our heuristics leverage the elasticity of cloud infrastructure at runtime, and we model the behavior of a virtualized commercial cloud Infrastructure as a Service (IaaS) environment here. The execution framework has access only to the virtualized cloud resources: virtualized CPU cores, virtual disks within a VM, and virtual network connectivity. There is no control over or knowledge of the actual VM placement within the data center and, consequently, the network connection behavior between the VMs. This implies different network profiles for different VM instances even of the same class.

The cloud environment consists of a set of VM resource classes $\mathcal{C} = \{C_1, C_2, ..., C_n\}$ that differ in the number of available CPU cores N, their rated core speed π, and their rated network bandwidth β. We assign atleast one dedicated core to each VM. For simplicity, we ignore memory and disk characteristics in the current model. Since CPU core speeds may vary across VM classes, we define the normalized processing power π_i of a resource class C_i's CPU core as the ratio of its processing power to that of a "standard" VM core, under ideal conditions. Naïvely, this may be the ratio of their CPU core clock speeds (e.g, $\frac{2.2\ GHz}{1.7\ GHz}$), but could also be the result of running application benchmarks on a standard VM and the VM of a particular resource class. Cloud providers such as Amazon also provide resource class ratings in the form of Elastic Compute Units (ECUs) that can be used. The processing requirements of a PE alternate is defined in terms of core-seconds (c) required to process a single message on the standard VM core (whose $\pi = 1$). Hence, the latency of a PE to process a message on a resource of class C_i can be obtained by scaling as $c_i = c/\pi_i$. In addition, each resource class is associated with a fixed hourly usage price ζ_i. We follow a costing model similar to existing cloud providers. The usage of a VM instance is rounded up to the nearest hourly boundary and the user is charged for the entire hour even if it is shut down before the hour ends.

$R(t) = \{r_1, r_2, ..., r_n\}$ is a set of all VMs that have been instantiated till time t. Each instance is described by the tuple $r_i = \langle C_j, t_{start}, t_{off} \rangle$, which is the resource class C_j to which the VM instance belongs, t_{start} is the time at which the instance was created and t_{off} is the time at which the instance was turned off. t_{off} is set to ∞ for an active VM. Total accumulated cost for the instance r_i at time t is then calculated as $\mu_i[t] = \lceil min(t_{off}, t) - t_{start})/60 \rceil \times \zeta_j$, where $min(t_{off}, t) - t_{start})$ is the duration for which the instance has been active.

Various studies, including our own, have shown that the performance of cloud VM instances is volatile, over time and across instances of the same class, including its processing power and network bandwidth [16, 18, 15]. To gage the current behavior of the virtualized cloud resource, we presume a monitoring framework that periodically and nonintrusively probes the performance of and faults in the VMs and their network connectivity using standard benchmarks.

The normalized processing power of a VM instance r_i's CPU core at time t thus monitored is given by $\pi_i(t)$, and the network latency and bandwidth between pairs of active VM instances r_i and r_j are $\lambda_{i \times j}(t)$ and $\beta_{i \times j}(t)$ respectively.

Hence, the processing latency for a PE alternate p_i^j at time t is a function of the current normalized processing power for the set of VM instances it is allocated at that time. Similarly, the bandwidth between two PE instances is the current bandwidth between the VM resources on which they are deployed. We assume in-memory message transfer if two PEs are collocated in the same VM instance, i.e., $\lambda_{i \times i} \to 0$ and $\beta_{i \times i} \to \infty$. In addition, during the deployment stage, we assume that the network bandwidth between two VMs is equal to the "rated values". However, during the runtime adaptation, we use the actual bandwidth, reported by the monitoring framework, which may change due to factors including collocation of VMs and data center traffic.

5. DYNAMIC DATAFLOW DEPLOYMENT & ADAPTATION

We outline here the overall approach to deployment and runtime adaptation of the dynamic dataflow on elastic cloud resources. The specific heuristics themselves are discussed in subsequent sections.

Fig. 1(a) shows a simple abstract dataflow with four PEs. E_1 and E_4 are the input and output PEs with just one alternate each, while PEs E_2 and E_3 have of two alternates each. Messages emitted on E_1's output port are duplicated to both E_2 and E_3 while E_4 interleaves the messages from the task parallel operations performed by E_2 and E_3. When this abstract dataflow is submitted for execution, the decisions on alternate selection and VM instance acquisition and allocation fall into two phases: at deployment time and at runtime. Deployment time heuristics (§ 7) determine the initial selection of the alternate for each PE, and the resources required by them based on estimated initial message data rates. Fig. 1(b) shows a sample concrete dataflow where the alternates have been selected, picking e_2^2 and e_3^1 for PEs E_2 and E_3, and the initial VM resources estimated for the four PEs (Fig. 1(c)).

At runtime, several instances of a PE (alternate) operate in a data parallel manner, with each CPU core allocated to the PE being able to operate on independent messages available on their input port. Incoming messages to a VM instance are buffered in a local queue before execution. The mechanics of these, while beyond the scope of this paper, are based on the $\mathcal{F}\ell o\varepsilon$ dataflow framework discussed elsewhere [34] [2]. PEs can also be allocated CPU cores that span multiple VMs; the logical operation of the dataflow is not affected by this though it does impact the performance of the PE due to the variations in processing power and network connectivity across the VMs. We assume incoming messages are load-balanced across the CPU cores, and releasing a VM instance migrates pending input messages in its buffer to remaining VMs hosting this PE (with network cost paid for the transfer). This also means that the PEs are stateless across messages, or can share state only between the instances running on the same VM. This allows us to allocate and de-allocate CPU cores on the same or different VMs rapidly without impacting the dataflow consistency. It also allows us to switch between alternates for a PE.

Based on the deployment time heuristics, VMs of particular resource classes are instantiated and the alternates of the dataflow initiated on them, following which the dataflow execution starts. This also initiates the monitoring probes on the VM instances to record their runtime performance, and measures the message data rates for the alternates. Periodically, we can perform runtime adaptation to respond to variability in incoming data rates and resource behavior using one of several controls that are available. These active decisions are based on runtime heuristics (§ 7) that can decide to pick a different alternate for a PE, change the resources allocated to an alternate within a VM (scale up/down) or on a new VM (scale out/in), or even decide to migrate an alternate from one VM instance to another. The VM acquisition and release are also tied to these decisions as it determines the actual cost paid to the cloud service provider.

This deployment and runtime adaptation model attempts to strike a balance between *simplicity* (e.g., statelessness and isolating PE instance on separate cores to minimize effect on each other), *reality of clouds* (e.g., costing at hour boundaries, network cost to migrate message buffers), and *user flexibility* (e.g., dynamic PEs, runtime VM and dataflow monitoring). This also allows us to formally define a meaningful yet tractable optimization problem for the deployment and runtime adaptation strategies that we introduce next.

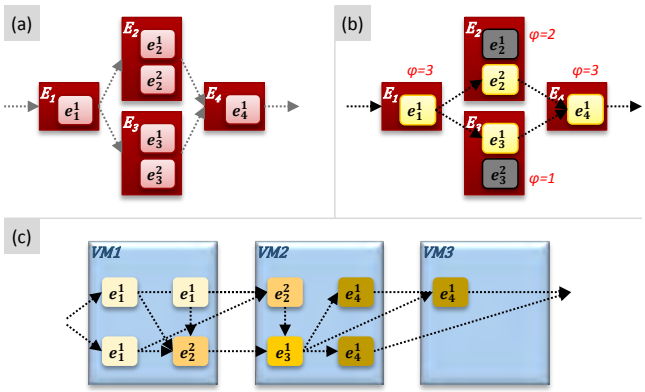


Figure 1: (a) An abstract dynamic dataflow application with possible alternates for PEs, (b) A concrete dynamic dataflow with alternates selected from possible combinations, and the initial resource requirements determined (c) A deployment of the concrete dataflow on VMs

6. PROBLEM FORMULATION

We formulate the optimization problem as a constrained utility maximization problem. The constraints to the function are defined in terms of the expected relative application throughput (Ω) (i.e., the throughput, on average, should not deviate from this requirement at the end of the optimization period). Similarly the utility is defined as a function of the normalized application value (Γ), the maximal cost they are willing to pay for the cloud resources, and the actual cost paid for the resources while achieving the constraint. The goal of the optimization problem is to maximize the utility (objective function) while minimizing the deviation from the defined constraint.

We define an optimization period T for which execution constraints are specified by the user and the utility needs to be maximized. This optimization period is divided into time intervals $T = \{t_0, t_1, ..., t_n\}$. While the period between the intervals could be of variable length, for the purpose of our theoretical analysis we assume that they are of equal length i.e., $t_{i+1} - t_i$ is a constant. The initial deployment decisions are made prior to t_0 and the the runtime decisions made at the beginning of each time interval.

At time t_0, an abstract dynamic dataflow D, along with $\mathcal{I}(t_0) = \{i_j(t_0)\}$, the estimated input data rates at each input PE $P_j \in I$ is given. During each subsequent time interval t, based on the monitoring during the previous time interval, we are also given the *observed* input data rates, $\mathcal{I}(t) = \{i_j(t)\}$; the set of active VM resource instances, $R(t) = \{r_1, r_2, ..., r_m\}$; the normalized processing power per

[2] $\mathcal{F}\ell o\varepsilon$ is available for download at https://github.com/usc-cloud/floe

core for each VM instance r_j, $\pi(t) = \{\pi_j(t)\}$; the network latency and bandwidth between pairs of active VM instances $r_i, r_j \in R(t)$, $\lambda(t) = \{\lambda_{i\times j}(t)\}$ and $\beta(t) = \{\beta_{i\times j}(t)\}$.

At any time interval t, we can calculate the *relative application throughput* ($\Omega(t)$) observed at time t using Def. 4, the *normalized application value* ($\Gamma(t)$) using Def. 3, and the cumulative monetary cost ($\mu[t]$) till time t.

The *average* relative application throughput over the optimization period T is given by $\Omega = \frac{\sum_{t\in T}\Omega(t)}{|T|}$, the *average* application value over period T is $\Gamma = \frac{\sum_{t\in T}\Gamma(t)}{|T|}$, and the *total* resource cost over period T is $\mu = \mu[t_n]$, where t_n is the last time interval in T.

We define the combined objective function which is to be maximized over the optimization period as a profit:

$$\Theta = \Gamma - \sigma \cdot \mu$$

where σ is a user-defined equivalence factor between cost and value given by

$$\sigma = \frac{MaxApplicationValue - MinApplicationValue}{AcceptableCostatMaxVal - AcceptableCostatMinVal}$$

Here, the max and min application values can be calculated from the alternates in the dataflow by picking the alternates with the best and worst values for each PE, while the user provides the acceptable costs to achieve these respective extremes. σ captures an intuitive linear function of the user's pricing expectation, and any cost below this linear function serves as a profit.

The goal is to maximize the objective function Θ while satisfying the constraint that the average relative throughput Ω over the optimization period meets a user-defined threshold $\Omega \geq \widehat{\Omega}$.

Θ can be maximized by choosing appropriate values for the following control parameters at each interval t in the optimization period: $A_i^j(t)$, which gives the active alternate j for the PE P_i; $N(t) = \{N^j(t)\}$, the number of VM instances in $R(t)$ belonging to resource class C_j; $\phi(t) = \{\phi_j(t)\}$ ϕ_t^j, the number of CPU cores available to P_j for data parallel execution; and $M(t) = \{M_{j\times k}(t)\}$, the mapping of a PE instance P_j to the actual resource instance r_k.

7. DEPLOYMENT AND ADAPTATION HEURISTICS

Optimally solving the objective function Θ with the Ω throughput constraint is *NP-Hard*. The proof is outside the scope of this paper, but we present an intuition here. $\Theta = \Gamma - \sigma \cdot \mu$ can be simplified by fixing either Γ or μ and optimizing the other. When the application value Γ is fixed, the problem then reduces to minimizing the execution cost μ of the dataflow, i.e., give the incoming data rate and fixed alternates, place PEs onto the VM instances of different resource classes such that the total cost of resources used is minimized while satisfying the $\Omega \leq \widehat{\Omega}$ constraint. This is similar to the *Variable-sized Bin Packing* problem [10], where given a set of objects and an infinite supply of different sized bins, the objective is to minimize the total space used to pack all the objects in the bins. This has been proven as an NP-Complete problem, and our problem is more complex than this reduced version. We thus propose heuristics to provide an approximate solution to the objective function.

Algorithm 1 Heuristic Algorithm for Initial Deployment

```
1: procedure InitialDeployment(Dataflow D)
                                          ▷ Alternate Selection Stage
2:     for Processing Element P ∈ D do
3:         for Alternate A ∈ P do
4:             μ ← GetCostOfAlternate(D, A, P)
5:             γ ← A.Value
6:             if γ/μ ≥ best then
7:                 best ← γ/μ
8:                 selected ← A
9:             end if
10:        end for
11:    end for
                                          ▷ Resource Allocation Stage
12:    VMClasses ← SortedByDecreasingSize()  ▷ size = N ×π
13:    while Ω ≤ Ω̂ do
14:        if (lastVM.isAvailable = false) then
15:            lastVM ← InitializeVM(VMClasses.First)
16:        end if
17:        P ← GetNextPE()
18:        CurrentAlloc ← AllocationOneCore(P, lastVM)
19:        Ω ← GetEstimatedThruput(D, CurrentAlloc)
20:    end while
21:    for Processing Element P ∈ D do
22:        if (IsOverProvisioned(P)) then
23:            RepackPE(P)
24:        end if
25:    end for
26:    RepackFreeVMs()      ▷ Repack PEs in VMs with free cores to smaller VMs
27: end procedure
```

We develop several heuristics to find an approximate solution to the optimization problem in near real time. While techniques such as integer programming and branch-and-bound have been used to optimally solve some NP-hard problems [41], such tractability does not adequately translate to low latency solutions that are critical to continuous adaptation decisions. The dynamic nature of the application and the infrastructure as well as the tightly-bound decision making interval means that fast heuristics are better suited than slow optimal solutions that may in any case become stale.

We distinguish between the initial deployment strategy and the runtime adaptation strategy. The initial deployment is based on the *estimated* data rate and the *expected* VM performance. At runtime, both of these may vary as observed by the monitoring framework, and the adaptation strategy changes the alternate selection and resource allocation. We propose an initial *local heuristic* that uses information available locally for a particular PE for decisions, and refine this to a *global heuristic* that also considers the impact on downstream PEs in the dataflow.

7.1 Initial Deployment Heuristics

The initial deployment algorithm (Alg. 1) is divided into two stages: alternate selection (lines 2-11) and resource allocation (lines 12-26).

The alternate selection stage ranks each alternate for a PE based on its value to cost ratio (line 4) and chooses the one with the highest ratio. Since we do not know the actual cost for the alternates until resource allocation, the local and global heuristics use cost functions for GetCostOfAlternate (Table 1). The local strategy calculates an alternate's cost based only on its processing requirement. The global

Function	Local Strategy	Global Strategy
GetCostOfAlternate	$A.cost$	$A.cost + S_i \times \sum successor.cost$

GetNextPE	**if** All PEs assigned **then**
	  return PE with lowest Ω_t^j ▷ bottleneck
	else
	  return Next PE in BFS
	end if	

RepackPE	N/A	Move PE to smallest VM big enough for required core-secs
Repack FreeVMs	N/A	Iterative Repacking [21]

Table 1: Functions used in Initial Deployment Strategies

strategy calculates the cost of the alternate as the sum of its processing needs and that of its downstream PEs – intuitively, if an upstream PE has more resources allocated, its output message rate increases and this has a cascading impact on the input rate of the succeeding PEs. A higher selectivity will have a higher impact on the successors since they will produce more output messages. This cost is calculated using a dynamic programming algorithm by traversing the dataflow graph in reverse BFS order rooted at the output PEs.

The resource selection stage follows a procedure similar to the *variable sized bin packing (VBP)* problem. For the initial deployment, in the absence of running VM instances, we assume that each instance from a resource class would behave ideally as per its rated performance. A generic VBP heuristic algorithm picks objects (PEs) in a certain order and puts each in the largest bin (VM instance), creating a new bin if required [13]. Based on some criterion, it chooses to "repack" the objects from the largest bin to smaller bins to avoid wasted space. Both, the order in which bins are chosen and the object selection for repacking affects the quality of the heuristic.

Both the local and global heuristics choose one PE (object) at a time using GetNextPE (line 17) and allocate it to the largest VM resource class, either available or newly instantiated (line 18). While the local strategy does not perform any repacking, the global strategy "repacks" individual PEs (line 23) and all PEs in a VM (line 26) using RepackPE and RepackFreeVMs (Table 1).

The intuition behind the GetNextPE function is to choose PEs in an order that not only reduces the spare capacity on a VM but also limits the message transfer latency between the PEs by collocating neighboring PEs in the dataflow within the same VM. We order the PEs based on a forward BFS traversal rooted at the input PEs, and allocate them resources in that order to increase the probability of collocating neighboring PEs. However, note that the CPU cores required for the individual PEs is not known in advance since the resource requirements depend on the current load which in turn depends on the resource requirements of the preceding PE. Hence, after assigning at least one CPU core to each PE (IncrementAllocation), the deployment algorithm chooses PEs in the order of largest bottlenecks in the dataflow, i.e., lowest relative PE throughputs (Ω). This ensures that PEs needing more resource are chosen first for allocation. This in turn may increase the input rate (and processing load) on the successive PEs, making them the bottlenecks. As a result, we take an iterative approach that incrementally allocates CPU cores to PEs until the throughput constraint is met. Since the resource allocation only impacts downstream PEs, this algorithm is bound to converge. We leave a theoretical proof to future work. Both the local and global strategies use this method to select the next PE for allocation.

The global strategy further performs two levels of repacking. After a solution is obtained using VM instances from just the largest resource class, it first moves the last instance for all the over-provisioned PEs to the smallest resource class large enough to accommodate that instance (best fit, using RepackPE). This may free up spare capacity on the initialized VMs. Hence, we again repack all the VM instances to minimize the wasted cores. Finally, an iterative repacking algorithm [21] is used to allocate cores for all the instances remaining on the last empty VM of the largest size (RepackVMs). During repacking we may sacrifice instance collocation in favor of reduced resource cost. The evaluation section however shows that this is an acceptable trade-off towards maximizing the objective function.

7.2 Runtime Adaptation Heuristics

The runtime adaptation kicks in after the deployed application has executed for a single time interval. Alg. 2 extends from the initial deployment algorithm but in addition considers the current state of the dataflow and cloud resources, available through monitoring, in adapting the alternate and resource selection. This gives a more accurate estimate of data rates (t_{i+1} is similar to t_i), and hence the resource requirements and the cost. As before, the algorithm is divided into two stages: alternate selection and resource allocation. However, unlike earlier, we do not run both the stages at the same time interval, but instead the former is run every m intervals and the latter is run every n intervals. We do so to keep a balance between application value which is decided by the alternate selection stage and the resource cost decided by the resource allocation stage.

During the alternate selection stage, we assume that the resource behavior will remain static for the next time interval. Hence, given the current data rate and resource performance, we calculate the resources needed for each alternate of a PE. As before, we use local and global variations of the strategy (line 5). We then create a list of "feasible" alternates for a given PE (lines 6-13), based on whether the current relative throughput is less or more than the expected throughput by a threshold. Finally, we sort the feasible alternates in the decreasing order of the value to cost ratio and select the first alternate which can be accommodated under the given available resources. The outcome of this phase is that the system either increases or decreases the overall value based on whether it is overprovisioned or underprovisioned respectively.

The resource re-deployment procedure is used to allocate or de-allocate resources as required to maintain the required relative throughput and to minimize the overall cost. If the average relative throughput thus far is less than $\widehat{\Omega}$, the algorithm proceeds similar to the initial deployment algorithm. It incrementally allocates additional resources to the bot-

Algorithm 2 Heuristic Algorithm for Runtime Adaptation

```
1: procedure AlternateReDeploy(DataflowD, Ω_t)   ▷ Ω_t is the observed relative throughput
2:     for PEP ∈ D do                            ▷ Alternate selection phase
3:         actual ← getAvailableResources()      ▷ Gets the current available resources
4:         for AlternateA ∈ P do
5:             actual ← actualResourceRequirements(A)
6:             if Ω_t ≤ Ω̂ − ϵ then
7:                 if actual ≤ activeAlternate.c then
8:                     feasible.add(A)
9:                 end if
10:            else if Ω_t ≥ Ω̂ + ϵ then
11:                if actual ≥ activeAlternate.c then
12:                    feasible.add(A)
13:                end if
14:            end if
15:        end for
16:        Sort(feasible)                        ▷ decreasing order of value/cost
17:        for feasible alternate A do
18:            if actual < available then
19:                SwitchAlternate(A)
20:                done
21:            end if
22:        end for
23:    end for
24: end procedure
1: procedure ResourceReDeploy(X)
2:     VMClasses ← SortedByDecreasingSize()      ▷ size = N ×π
3:     if Ω_t ≤ Ω̂ then
4:         Same procedure as initial deployment
5:     else if Ω_t ≥ Ω̂ then
```

8. EVALUATION

We evaluate the proposed heuristics through extensive simulations using an IaaS cloud simulator that uses performance traces from a real deployment on a medium sized private cloud infrastructure. Our *IaaSSimulator* is similar, in principle, to CloudSim [6] in that it allows users to test the expected performance and cost characteristics of their applications on a cloud environment without the (prohibitive) cost and effort of an actual deployment. However, unlike CloudSim, we provide an abstraction at the IaaS layer and hide the details of the underlying data center architecture. In addition, to mimic the cloud behavior – especially the observed performance variations – IaaSSimulator supports infrastructure dynamism and also allows replaying performance traces gathered from real cloud systems to accurately simulate the infrastructure characteristics [3].

8.1 Experiments

The experimental setup consists of a small abstract dynamic dataflow with several alternates as shown in Fig. 1. Even though we use a small abstract dataflow, it is scaled up to 10's of alternates and 100's of VMs to accommodate high and variable data rates that demonstrates scalability of the proposed heuristics. We use the same virtual machine instance types as provided by the AWS cloud provider with similar performance ratings and on-demand pricing per hour. For simplicity, during initial deployment, we assume an average network bandwidth between VMs of any types to be 100 Mbps.

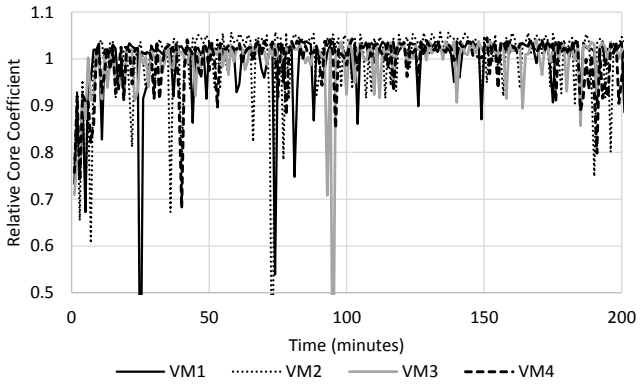


Figure 2: Variations in VM CPU performance in a private IaaS cloud

We simulate infrastructure variability by replaying the performance traces obtained from running periodic micro-benchmarks on 20 VMs in the FutureGrid Eucalyptus 2.0 private cloud for 26 days. Fig. 2 plots a representative sample of metrics from four VMs, showing the CPU performance

[3]IaaSSimulator is available for download at https://github.com/usc-cloud/IaaSSimulator

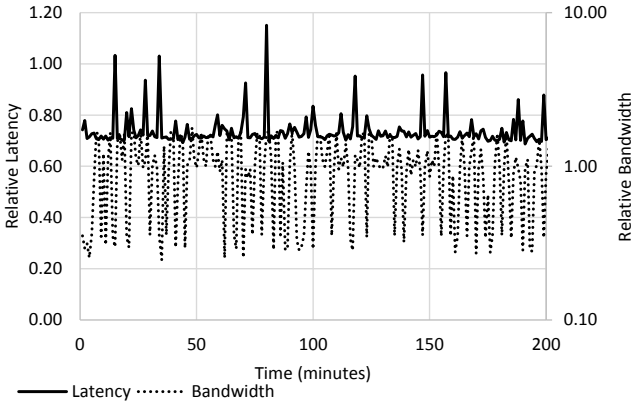
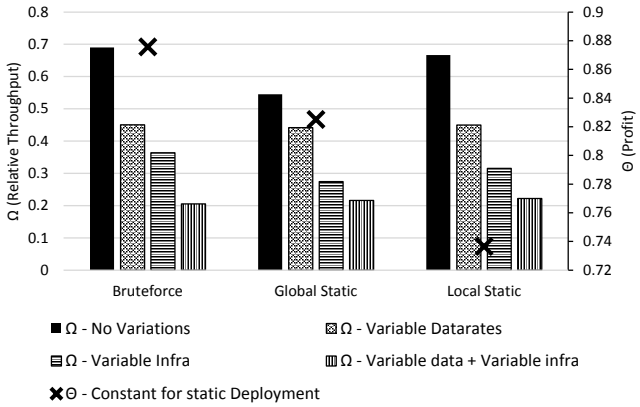


Figure 3: Variations in network performance between a pair of VMs in a private IaaS cloud

variability for those VMs over a four day period and the re tive deviation of CPU performance from its mean. Similar Fig. 3 shows variations in network latency and bandwid availability between a pair of VMs over the same time p riod. Our experiments on the FutureGrid private cloud cc form with other such studies [16] that illustrate high va ations in the performance of IaaS clouds. For individu experimental runs, we assign a random time period fr the traces for each active VM to replay. We then multip that coefficient with the rated performance of the active V to obtain its instantaneous runtime performance.

To evaluate the efficacy of the proposed heuristics, compare them against several other implementations inclu ing a static bruteforce optimal deployment for small grap (that assumes no variations) as well as a static deployme using the local and the global heuristic. To specifically ev uate the benefits of using our notion of dynamic applic tions, we also compare our heuristics against a simplifi version which ignores alternate selection as an optimizati

relative throughput (Ω) on the Y axis for different scenarios at a fixed data rate of 5 msgs/sec. With no (data or infrastructure) variability, the bruteforce algorithm performs the best (i.e. it always satisfies the Ω constraint and gives the highest Θ value), followed by the local heuristic and then the global heuristic. For small data rates, the local heuristic performs slightly better in terms of Ω because global heuristic involves repacking the empty virtual machines which sacrifices PE collocation. However, as we see later, this issue is mitigated by the continuous version of the global heuristic. As we introduce variability in data, infrastructure or both, the performance of even the bruteforce static deployment decreases drastically and none of the algorithms are able to satisfy the Ω constraint, though the Θ value remains the same. This proves the need for continuous redeployment strategies. Besides the brute-force algorithm which takes prohibitively long to find a solution for higher data rates, the performance of both the static deployment algorithms (i.e., local static and global static) decreases as the data rate increases, even with no variations (Fig. 5). This



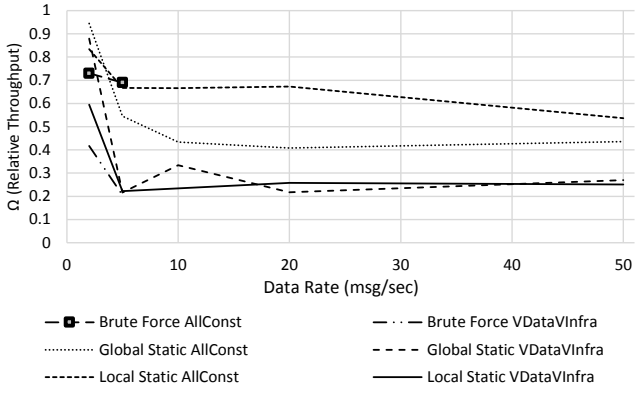


Figure 5: Effect of data rates on relative throughput, for static deployments

Next, we compare the local heuristic against the global

PEs and avoids re-deploym

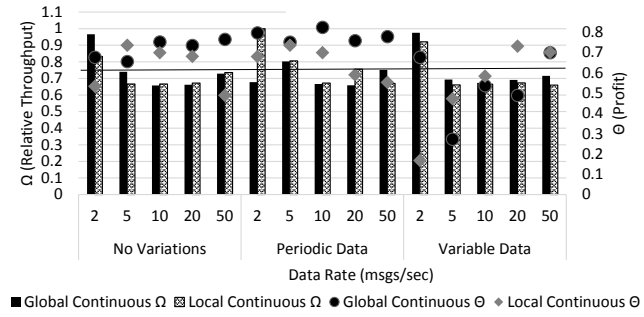


Figure 6: Performance of Local vs. Global Heuristic for stable infrastructure and variation in data rate

Fig. 7 compares the two heuristics at the other extreme, where the data rate remains constant but there are variations in the infrastructure performance. Here, as well, we observe both the local and the global heuristics meeting the Ω constraint (within an $\epsilon \leq 0.05$ threshold) for the range of data rates. As before, we also see that the global heuristic leads to higher Θ value for high data rates, while the local heuristic fairs well for smaller data rates ($\leq 5msg/s$). Further Fig. 8, compares the two under both data and infrastructure variability. We see similar trends in this graph too, with both the heuristics satisfying the Ω constraint and the global heuristic significantly out-performing the local heuristic at higher data rates (leading to a $\sim$60% reduction in the overall dollar cost). However, the local heuristic does continue to perform much better for small data rates

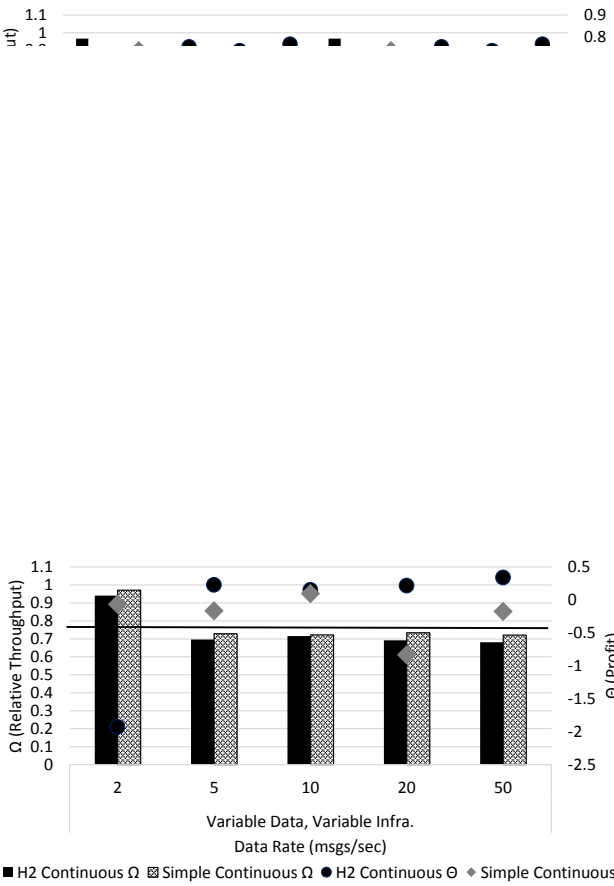


Figure 8: Performance of Local vs. Global Heuristic for variations in both data rates and infrastructure

In summary, so far we have seen that the static algorithms (including static brute-force optimal) perform poorly in the presence of any form of (data or infrastructure) variability. While the global heuristic performs better than the local heuristic in presence of either or both forms of variability at high data rates, the local heuristic does better with low data rates. Both the heuristics take advantage of the alternates provided by the dynamic tasks to increase the overall Θ value. Finally, we characterize the unique advantage offered by the alternates in the dynamic applications. We compare variations of the proposed heuristics while enabling or disabling application dynamism (i.e. the alternate selection stage) and hence compare the selective advantage achieved through application dynamism.

Fig. 9 compares the dollar amount spent by various heuristics over a period of 10 hours for various data rates. The heuristics we compare are global, global without application dynamism, local, and local without application dynamism. As seen before, global performs the best with minimum dollar cost for higher data rates and the local heuristic does so for smaller data rates. We also see that the global heuristic without application dynamism costs more than global in every single case. On an average global requires about 15% less dollar amount than the global without application dynamism. The global heuristic out-performs the local heuristic without application dynamism and allows for a savings of upto 70%.

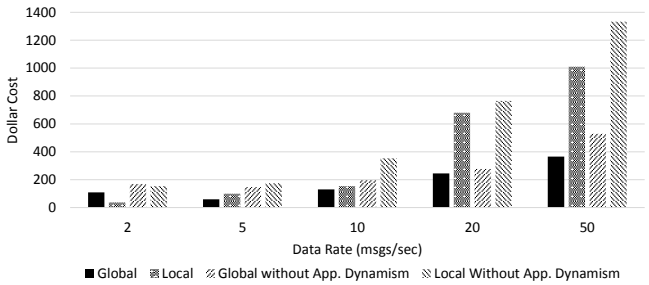


Figure 9: Dollar cost benefit of application dynamism with continuous re-deployment

9. CONCLUSION

In this paper, we have empirically demonstrated the need for online monitoring and re-deployment of continuous dataflow applications to meet their QoS constraints in the presence of either data or infrastructure variability or both. We further introduced the notion of dynamic dataflows with support for alternate implementations for the tasks in the dataflow. This not only gives user the flexibility in terms of application composition, but also gives an additional point of control which can be leveraged to meet the application constraints while maximizing the application value. Our experimental results show that the continuous re-deployment heuristic which makes use of application dynamism can reduce the execution cost by up to 15% on clouds while also meeting the QoS constraints.

As future work, we propose to extend the concept of dynamic tasks to dynamic paths. This will further allow for alternate implementations at coarser granularities, such as a subset of the application graph, and provide end user with more sophisticated and intuitive controls. In addition, we also plan to investigate the application of dynamic tasks to support enhanced fault tolerance and recovery mechanisms in continuous dataflow.

10. ACKNOWLEDGMENTS

This work was supported by grants from the DARPA XDATA and NSF CiC programs (CCF-1048311). We thank FutureGrid for resources provided under NSF Award 0910812.

11. REFERENCES

[1] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*. ACM, 2006.

[2] S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[3] R. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, and Y. Simmhan. The trident scientific workflow workbench. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, 2008.

[4] A. Barker and J. Van Hemert. Scientific workflow: a survey and research directions. In *Parallel Processing and Applied Mathematics*, pages 746–753. 2008.

[5] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. Ibm infosphere streams for scalable, real-time, intelligent transportation services. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[6] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[7] S. Carlsen. Action port model: A mixed paradigm conceptual workflow modeling language. In *Cooperative Information Systems, 1998. Proceedings. 3rd IFCIS International Conference on*, 1998.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003.

[9] L. Douglas. 3d data management: Controlling data volume, velocity, and variety. Technical report, Gartner, 2001.

[10] D. K. Friesen and M. A. Langston. Variable sized bin packing. *SIAM journal on computing*, 15(1):222–230, 1986.

[11] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *Services Computing, IEEE Transactions on*, 3(3):223–235, 2010.

[12] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. 2012.

[13] M. Haouari and M. Serairi. Heuristics for the variable sized bin-packing problem. *Computers & Operations Research*, 36(10):2877–2884, 2009.

[14] W. Hummer, P. Leitner, and S. Dustdar. Ws-aggregation: distributed aggregation of web services data. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011.

[15] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, 2011.

[16] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*.

[17] A. Ishii and T. Suzumura. Elastic stream computing with clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, july 2011.

[18] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168. IEEE, 2010.

[19] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl. Qos aggregation for web service composition using

workflow patterns. In *Enterprise distributed object computing conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, pages 149–159, 2004.

[20] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Scientific workflow applications on amazon ec2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66. IEEE, 2009.

[21] J. Kang and S. Park. Algorithms for the variable sized bin packing problem. *European Journal of Operational Research*, 147(2):365–372, 2003.

[22] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[23] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and distributed Computing*, 59(2):107–131, 1999.

[24] E. M. Maximilien and M. P. Singh. A framework and ontology for dynamic web services selection. *Internet Computing, IEEE*, 8(5):84–93, 2004.

[25] P. Neophytou, P. Chrysanthis, and A. Labrinidis. Confluence: Implementation and application design. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2011 7th International Conference on*, oct. 2011.

[26] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177.

[27] G. J. Nutt. The evolution towards flexible workflow systems. *Distributed Systems Engineering*, 3(4), 1996.

[28] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, et al. Nova: continuous pig/hadoop workflows. In *international conference on Management of data*, 2011.

[29] S. Pallickara, J. Ekanayake, and G. Fox. Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009.

[30] R. Prodan and T. Fahringer. Dynamic scheduling of scientific workflow applications on the grid: a case study. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 687–694. ACM, 2005.

[31] S. W. Sadiq, M. E. Orlowska, and W. Sadiq. Specification and validation of process constraints for flexible workflows. *Information Systems*, 30(5):349–378, 2005.

[32] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*.

[33] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 25–36. IEEE, 2003.

[34] Y. Simmhan, A. Kumbhare, and C. Wickramachari. Floe: A dynamic, continuous dataflow framework for elastic clouds. Technical report, USC, 2013. http://ceng.usc.edu/šimmhan/pubs/simmhan-usctr-2013.pdf.

[35] I. J. Taylor, E. Deelman, and D. B. Gannon. Workflows for e-science: scientific workflows for grids.

[36] R. Tolosana-Calasanz, J. At'ngel Bañares, C. Pham, and O. Rana. End-to-end qos on shared clouds for highly dynamic, large-scale sensing data streams. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 904–911. IEEE, 2012.

[37] H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[38] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.

[39] J. Wainer and F. de Lima Bezerra. Constraint-based flexible workflows. In *Groupware: Design, Implementation, and Use*, pages 151–158. 2003.

[40] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22, 1997.

[41] G. J. Woeginger. Exact algorithms for np-hard problems: A survey. In *Combinatorial Optimization—Eureka, You Shrink!*, pages 185–207. Springer, 2003.

[42] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *Sigmod Record*, 34(3):44, 2005.

[43] J. Yu and R. Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Scientific Programming*, 14(3):217–230, 2006.

[44] J. Yu, R. Buyya, and C. K. Tham. Cost-based scheduling of scientific workflow applications on utility grids. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8–pp. IEEE, 2005.

[45] T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web*, 1(1):6, 2007.

[46] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[47] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10, 2012.

[48] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 54–71. Springer, 2006.

GoFFish: A Sub-Graph Centric Framework for Large-Scale Graph Analytics

Yogesh Simmhan[1], Alok Kumbhare[2], Charith Wickramaarachchi[2], Soonil Nagarkar[2], Santosh Ravi[2], Cauligi Raghavendra[2], and Viktor Prasanna[2]

[1] Indian Institute of Science, Bangalore India
[2] University of Southern California, Los Angeles USA
simmhan@serc.iisc.in, {kumbhare, cwickram, snagarka, sathyavi, raghu, prasanna}@usc.edu

Abstract. Vertex centric models for large scale graph processing are gaining traction due to their simple distributed programming abstraction. However, pure vertex centric algorithms under-perform due to large communication overheads and slow iterative convergence. We introduce *GoFFish* a scalable sub-graph centric framework co-designed with a distributed persistent graph storage for large scale graph analytics on commodity clusters, offering the added natural flexibility of shared memory sub-graph computation. We map Connected Components, SSSP and PageRank algorithms to this model and empirically analyze them for several real world graphs, demonstrating *orders of magnitude improvements*, in some cases, compared to Apache Giraph's vertex centric framework.

1 Introduction

One defining characteristic of complexity in "Big Data" is the intrinsic interconnectedness, endemic to novel applications in both the Internet of Things and Social Networks. Such graph datasets offer unique challenges to scalable analysis, even as they are becoming pervasive. There has been significant work on parallel algorithms and frameworks for large graph applications on HPC clusters [1][3], massively multi-threaded architectures [2], and GPUs [3]. Our focus here, however, is on leveraging *commodity hardware* for scaling graph analytics. Such distributed infrastructure, including Clouds, have democratized resource access, as evidenced by popular programming frameworks like MapReduce. While MapReduce's tuple-based approach is ill-suited for many graph applications [4], recent *vertex-centric programming abstractions* [5,6], like Google's Pregel and its open-source version, Apache Giraph [7], marry the ease of specifying a uniform logic for each vertex with a *Bulk Synchronous Parallel (BSP)* execution model to scale [4]. Independent vertex executions in a distributed environment are interleaved with synchronized message exchanges across them to form iterative *supersteps*.

However, there are short-comings to this approach. (1) Defining an individual vertex's logic forces costly messaging even within vertices in one partition. (2) Porting shared memory graph algorithms to efficient vertex centric ones can be non-trivial. (3)

[3] The Graph 500 List, http://www.graph500.org
[4] Scaling Apache Giraph to a Trillion Edges, *Facebook Engineering*, http://on.fb.me/1czMarU

The programming abstraction is decoupled from the data layout on disk, causing I/O penalties at initialization and runtime. A recent work [6] identified the opportunity of leveraging shared memory algorithms within a *partition*, but relaxed the programming model to operate on a whole partition, without guarantees of *sub-graph connectivity* within a partition or implicit use of *concurrency* across sub-graphs.

In this paper, (1) we propose a sub-graph centric programming abstraction, *Gopher*, for performing distributed graph analytics. This balances the flexibility of reusing shared-memory graph algorithms over *connected* sub-graphs while leveraging sub-graph concurrency within a machine, and distributed scaling using BSP. (2) We couple this abstraction with an efficient distributed storage, *GoFS*. GoFS stores partitioned graphs across distributed hosts while coalescing connected sub-graphs within partitions, to optimize sub-graph centric access patterns. Gopher and GoFS are co-designed as part of the *GoFFish* graph analytics framework, and are empirically shown here to scale for common graph algorithms on real world graphs, in comparison with Apache Giraph.

2 Background and Related Work

There is a large body of work on parallel graph processing [8] for HPC clusters [1], massively multi-threaded architectures [2], and GPGPUs [3], often tuned for specific algorithms and architectures [9]. For e.g., the STAPL Parallel Graph Library (SGL) [10] offers diverse graph data abstractions, and can express level-synchronous vertex-centric BSP and coarse-grained algorithms over sub-graphs, but not necessarily a sub-graph centric BSP pattern as we propose. SGL uses STAPL for parallel execution using OpenMP and MPI, which scales on HPC hardware but not on commodity clusters with punitive network latencies. Frameworks for commodity hardware trade performance in favor of scalability and accessibility – ease of use, resource access, and programming.

The popularity of MapReduce for large scale data analysis has extended to graph data as well, with research techniques to scale it to peta-byte graphs for some algorithms [11]. But the tuple-based approach of MapReduce makes it unnatural for graph algorithms, often requiring new programming constructs [12], platform tuning [4], and repeated reads/writes of the entire graph to disk. Google's recent Pregel [5] model uses iterative supersteps of vertex centric logic executed in a BSP model [13]. Here, users implement a `Compute` method for a single vertex, with access to its value(s) and outgoing edge list. `Compute` is executed concurrently for each vertex and can emit messages to neighboring (or discovered) vertices. Generated messages are delivered in bulk and available to the target vertices only after all `Compute`s complete, forming one barriered *superstep*. Iterative level-synchronized supersteps interleave computation with message passing. The vertices can `VoteToHalt` in their `Compute` method at any superstep; any vertex that has voted to halt is not invoked in the next superstep unless it has input messages. The application terminates when all vertices vote to halt and there are no new input messages available. Pseudocode to find the maximal vertex using this model is shown in Alg. 1. Apache Giraph [7] is an open source implementation of Google's Pregel, and alternatives such as Hama [14], Pregel.NET [15] and GPS [16] also exist.

Despite the programming elegance, there are key scalability bottlenecks in Pregel: (1) the number of messages exchanged between vertices, and (2) the number of synchronized supersteps required for completion. Message passing is done either in-memory

(for co-located vertices) or over the network, while barrier synchronization across distributed vertices is centrally coordinated. This often makes them communication bound on commodity hardware, despite use of `Combiners` [5] for message aggregation. Secondly, the default hashing of vertices to machines exacerbates this though better partitioning shows mixed results [16,6]. Even in-memory message passing causes memory pressure [15]. For e.g., GPS [16] performs dynamic partition balancing and replication on Pregel and our prior work [15] used a swathe-based scheduling to amortize messaging overhead. But these engineering solutions do not address key limitations of the abstraction, which also leads to large number of supersteps to converge for vertex centric algorithms (e.g.$\sim$30 for PageRank, or graph diameter for Max Vertex).

Recently, Tian, et al. [6] recognized these limitations and propose a *partition centric* variant, *Giraph++*. Here, users' `Compute` method has access to all vertices and edges in a partition on a machine, and can define algorithms that operate on a whole partition within a superstep before passing messages from *boundary vertices* of the partition to neighboring vertices. They also partition the graph to minimize average *ncuts*. Such local compute on the coarse partition can reduce the number of messages exchanged and supersteps taken, just as for us. But this approach falls short on several counts. (1) Though the partitions are called "sub-graphs" in Giraph++, these sub-graphs are not connected components. This limits the use of shared-memory graph algorithms that operate on connected graphs, and can lead to a suboptimal algorithm operating collectively on hundreds of sub-graphs in a partition. This also puts the onus on the user to leverage concurrency across sub-graphs in a partition. Our proposed abstraction and execution model *a priori* identifies *sub-graphs as locally connected components*; users define their `Compute` method on these. Our engine also automatically executes sub-graphs in parallel on the local machine. (2) Their `Compute` can send messages to only boundary vertices. We also allow messages to be sent to sub-graphs, fully exploiting the abstraction. (3) Our distributed graph storage is designed for sub-graph access patterns to offer data loading efficiencies, and extends beyond just prior graph partitioning.

Algorithm 1 Max Vertex Value using Vertex Centric Model

```
1:  procedure COMPUTE(Vertex myVertex, Iterator⟨Message⟩ M)
2:      hasChanged = (superstep == 1) ? true : false
3:      while M.hasNext do          ▶ Update to max message value
4:          Message m ← M.next
5:          if m.value > myVertex.value then
6:              myVertex.value ← m.value
7:              hasChanged = true
8:      if hasChanged then          ▶ Send message to neighbors
9:          SENDTOALLNEIGHBORS(myVertex.value)
10:     else
11:         VOTETOHALT( )
```

Distributed GraphLab [17] is another popular graph programming abstraction, optimized for local dependencies observed in data mining algorithms. GraphLab too uses an iterative computing model based on vertex-centric logic, but allows asynchronous execution with access to vertex neighbors. Unlike Pregel, it does not support graph mutations. There are other distributed graph processing systems such as Trinity [18]

that offer a shared memory abstraction in a distributed memory infrastructure. Here, algorithms use both message passing and a distributed address space called *memory cloud*. However, this assumes large memory machines with high speed interconnects. We focus on commodity cluster and do not make such hardware assumptions.

3 Sub-graph Centric Programming Abstraction

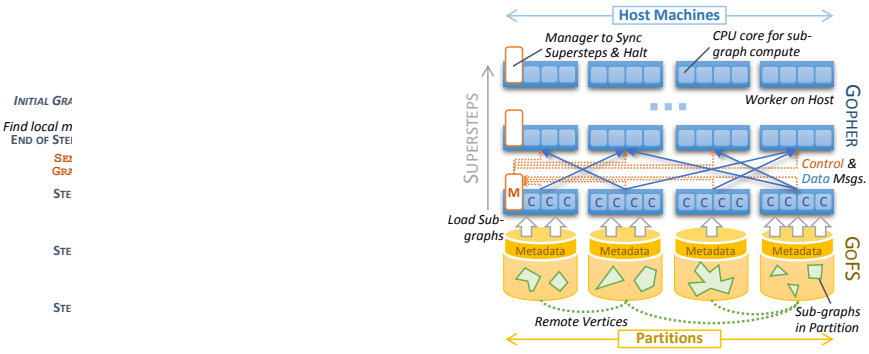
We propose a sub-graph centric programming abstraction that targets the deficiencies of a vertex centric BSP model. We operate in a distributed environment where the graph is $k - way$ partitioned over its vertices across k machines. We define a **sub-graph** as a connected component within a partition of an undirected graph; they are weakly connected if the graph is directed. The Fig. 1(a) shows two partitions with three sub-graphs. Two sub-graphs do not share the same vertex but can have *remote edges* that connect their vertices (dotted edges in Fig. 1(a)), as long as the sub-graphs are on different partitions. If two sub-graphs on the same partition share a *local edge* (solid edges), by definition they are merged into a single sub-graph. A partition can have one or more sub-graphs and the set of all sub-graphs forms the complete graph. Specific partitioning approaches are discussed later, and each machine holds one partition. Sub-graphs behave like "meta vertices" with remote edges connecting them across partitions.

Formally, let $P_i = \{\mathbb{V}_i, \mathbb{E}_i\}$ be a graph partition i where V_i and E_i are the set of vertices and edges in the partition. We define a *sub-graph* S in P_i as $S = \{V, E, R | v \in V \Rightarrow v \in \mathbb{V}_i; e \in E \Rightarrow e \in \mathbb{E}_i; r \in R \Rightarrow r \notin \mathbb{V}_i; \forall u, v \in V \exists$ an undirected path between u, v; and $\forall r \in R \; \exists v \in V$ s.t. $e = \langle v, r \rangle \in E\}$ where V is a set of local vertices, E is a set of edges and R is a set of remote vertices.

Each sub-graph is treated as an independent unit of computation within a BSP superstep. Users implement the following method signature:

```
Compute(Subgraph , Iterator<Message>)
```

The C
and edge



(b)

Fig. 1: (a) Sub-graph centric Max Vertex using Alg. 2. Dashed arrows show messages passed. (b) Sub-graph centric data access from GoFS and BSP execution by Gopher.

superstep and accumulate values of the sub-graph or update values for the local vertices and edges. Different sub-graphs communicate by message passing, with messages exchanged at synchronized superstep boundaries in a BSP model. Several methods enable this messaging. Algorithms often start by sending messages to neighboring sub-graphs.

```
SendToAllSubGraphNeighbors(Message)
```

As other sub-graphs are discovered across supersteps, two other methods are useful:

```
SendToSubGraph(SubGraphID, Message)
SendToSubGraphVertex(SubGraphID, VertexID, Message)
```

We allow a (costly) broadcast to all sub-graphs, though it should be used sparingly.

```
SendToAllSubGraphs(Message)
```

As with Pregel, the `Compute` method can invoke `VoteToHalt`. The application terminates when all sub-graphs have halted and there are no new input messages.

Alg. 2 presents the sub-graph centric version for finding the maximum vertex value in a graph. Fig. 1(a) illustrates its execution. The `Compute` method operates on a sub-graph *mySG*. Lines 2–6 are executed only for the first superstep, where each sub-graph's value is initialized to largest of its vertices. Subsequently, the algorithm is similar to the vertex centric version: we send the sub-graph's value to its neighboring sub-graphs and update the sub-graph's value to the highest value received, halting when there is no further change. At the end, each sub-graph has the value of the largest vertex.

Algorithm 2 Max Vertex using Sub-Graph Centric Model

```
 1: procedure COMPUTE(SubGraph mySG, Iterator⟨Message⟩ M)
 2:     if superstep = 1 then            ▷ Find local max in subgraph
 3:         mySG.value ← −∞
 4:         for all Vertex myVertex in mySG.vertices do
 5:             if mySG.value < myVertex.value then
 6:                 mySG.value ← myVertex.value
 7:     hasChanged = (superstep == 1) ? true : false
 8:     while M.hasNext do
 9:         Message m ← M.next
10:         if m.value > mySG.value then
11:             mySG.value ← m.value
12:             hasChanged = true
13:     if hasChanged then
14:         SENDTOALLSUBGRAPHNEIGHBORS(mySG.value)
15:     else
16:         VOTETOHALT( )
```

Compared to the vertex centric algorithm, the sub-graph centric version reduces the number of supersteps taken since the largest value discovered at any superstep propagates through the entire sub-graph in the same superstep. For e.g., for the graph in Fig. 1(a), the vertex centric approach takes 7 supersteps while we use 4. Also, this reduces the cumulative number of messages exchanged on the network. In the worst case, when a sub-graph is trivial (has one vertex), we degenerate to a vertex centric model.

Benefits. Our elegant extension of the vertex centric model offers three key benefits.

1) Messages Exchanged. Access to the entire sub-graph enables the application to make a significant progress within each superstep while reducing costly message exchanges that cause network overhead, disk buffering or memory pressure, between supersteps. While Pregel allows `Combiners` per worker, they operate after messages are generated while we preclude message generation. Giraph++ has similar advantages, but no better; as sub-graphs in a partition are disconnected, they do not exchange messages.

2) Number of Supersteps. Depending on the algorithm, a sub-graph centric model can reduce the number of supersteps required compared to Pregel, thereby limiting synchronization overheads. Also, the time taken by a superstep is based on its slowest sub-graph, with a wide distribution seen across sub-graphs [15]. Reducing the supersteps mitigates this skew. For traversal algorithms, the number of supersteps is a function of the graph *diameter*. Using a sub-graph centric model, this reduces to the diameter of the meta-graph where the sub-graphs form meta-vertices. In the best case (a linear chain), the number of supersteps can reduce proportional to the number vertices in a sub-graph, while for a trivial sub-graph, it is no worse. These benefits translate to Giraph++ too.

3) Reuse of Single-machine Algorithms. Lastly, our approach allows direct reuse of efficient shared-memory graph algorithms on a sub-graph, while using a BSP model across supersteps. The change from a simple vertex-centric approach is incremental, but the performance improvement stark. e.g. In Alg. 2, but for the shaded lines 2–6 which operates on the whole sub-graph, other lines are similar to Alg. 1. This has two benefits relative to Pregel and Giraph++: (1) We can leverage optimal single machines algorithms for sub-graphs, even leveraging GPGPU accelerators, with the added guarantee that the *sub-graphs are connected* (unlike Giraph++). This ensures traversals reach all sub-graph vertices and avoids testing every vertex in the partition independently. Second, when a partition has multiple sub-graphs, as is often, we can exploit concurrency across them automatically. While the degree of parallelism is not as high as vertex centric (Pregel), it is better than treating the partition as one computation unit (Giraph++).

4 Architecture

GoFFish is a scalable software framework for storing graphs, and composing and executing graph analytics [5]. A *Graph oriented File System (GoFS)* and *Gopher execution engine* are co-designed *ab initio* to ensure efficient distributed storage for sub-graph centric data access patterns. The design choices target commodity or virtualized hardware with Ethernet and spinning disks. GoFFish is implemented in Java.

GoFS Distributed Graph Store. GoFS is a distributed store for partitioning, storing and accessing graph datasets across hosts in a cluster. Graphs can have both a *topology* and *attributes* associated with each vertex and edge. The former is an adjacency list of uniquely labeled vertices and (directed or undirected) edges connecting them. Attributes are a list of name-value pairs with a schema provided for typing. Input graphs are partitioned across hosts, one partition per machine, using the METIS tool [19] to balance vertices per partition and minimize edge cuts (Fig. 1(b)).

GoFS uses a sub-graph oriented model for mapping the partition's content to *slice files*, which form units of storage on the local file system. We identify all sub-graphs

[5] `https://github.com/usc-cloud/goffish`

in the partition – components that are (weakly) connected through local edges, and a partition with n vertices can have between 1 to n sub-graphs. Each sub-graph maps to one *topology slice* that contains local vertices, local edges and remote edges, with references to partitions holding the destination remote vertex, and several *attribute slices* that hold their names and values. We use *Kryo* [6] for compact object storage on disk.

GoFS is a *write once-read many* scalable data store rather than a database with rich query support. The GoFS Java API allows clients to access a graph's metadata, attribute schema and sub-graphs present in the local partition. Specific sub-graphs and select attributes can be loaded into memory and traversed. Remote edges in a sub-graph resolve to a remote partition/sub-graph/vertex ID that can be used to send messages to.

Gopher Sub-graph Centric Framework. The Gopher programming framework implements our proposed sub-graph centric abstractions, and executes them using the *Floe* [20] dataflow engine on a Cloud or cluster in conjunction with GoFS. Users implement their algorithm in Java within a `Compute` method where they get access to a local sub-graph object and data messages from the previous superstep. They use `Send*` methods to send message to the remote sub-graphs in the next superstep and can `VoteToHalt()`. The same `Compute` logic is executed on every sub-graph in the graph, for each superstep.

The Gopher framework has a *compute worker* running on each machine and a *manager* on one machine. The workers initially load all local sub-graphs for that graph into memory from GoFS. For every superstep, the worker uses a multi-core-optimized task pool to invoke the `Compute` on each sub-graph, transparently leveraging concurrency within a partition. `Send*` messages are resolved by GoFS to a remote partition and host. The worker aggregates messages destined for the same host and sends them asynchronously to the remote worker while the compute progresses.

Once the `Compute` for all sub-graphs in a partition complete, the worker flushes pending messages to remote workers and signals the manager. Once the manager receives signals from all workers, it broadcasts a *resume* signal to the workers to start their next superstep and operate on input messages from the previous superstep. `Compute` is stateful for each sub-graph; so local variables are retained across supersteps. When a worker does not have to call `Compute` for any of its sub-graphs in a superstep, because all voted to halt *and* have no input messages, it sends a *ready to halt* signal to the manager. When all workers are ready to halt, the manager terminates the application.

Storage-Compute Co-design. Co-designing data layout and execution models is beneficial, as seen with Hadoop and HDFS. GoFFish uses sub-graphs as a logical unit of storage and computation; hence our data store first partitions the graph followed by sub-graph discovery. Partitioning minimizes network costs when loading sub-graphs into Gopher. We use existing partitioning tools (METIS) to balance vertices and minimize edge cuts. Ideally, we should also balance the number of sub-graphs per partition and ensure uniform size to reduce compute skew in a superstep. Further, having multiple sub-graphs in a partition can leverage the concurrency across sub-graphs. Such schemes are for future work. We also balance the disk latency against bytes read by slicing sub-graphs into topology and attributes files. For e.g. a graph with 10 edge attributes that uses only the *weight* attribute in an algorithm needs to load only one attribute slice.

[6] Kryo serialization framework, `https://github.com/EsotericSoftware/kryo`

5 Evaluation of Sub-graph Centric Algorithms on GoFFish

We present and evaluate several sub-graph centric versions of common graph algorithms, both to illustrate the utility of our abstraction and the performance of GoFFish. We comparatively evaluate against Apache Giraph, a popular implementation of Pregel's vertex centric model, that uses HDFS. We use the latest development version of Giraph, at the time of writing, which includes recent performance enhancements. Sub-graph centric Gopher and vertex centric Giraph algorithms are implemented for: *Connected Components*, *Single Source Shortest Path (SSSP)* and *PageRank*.

Experimental Setup and Datasets. We run these experiments on a modest cluster of 12 nodes, each with an 8-core Intel Xeon CPU, 16 GB RAM, 1 TB SATA HDD, and connected by Gigabit Ethernet. This is representative of commodity clusters or Cloud VMs accessible to the long tail of science rather than HPC users. Both Giraph and GoFFish are deployed on all nodes, and use Java 7 JRE for 64 bit Ubuntu Linux. The GoFFish manager runs on one node.

We choose diverse real world graphs (Table 1): California road network (**RN**), Internet topology from traceroute statistics (**TR**)), and LiveJournal social network (**LJ**). RN is a small, sparse network with a small and even edge degree distribution, and a large diameter. LJ is dense, with powerlaw edge degrees and a small diameter. TR has a powerlaw edge degree, with a few highly connected vertices. Unless otherwise stated, we report average values over three runs each for each experiment.

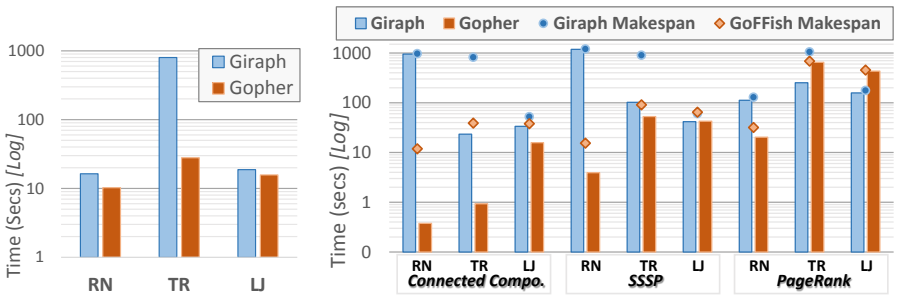
Table 1: Characteristics of graph datasets used in evaluation

Graph	*Vertices*	*Edges*	*Diameter*	*WCC*
RN	1,965,206	2,766,607	849	2,638
TR	19,442,778	22,782,842	25	1
LJ	4,847,571	68,475,391	10	1,877

Summary Results. We compare the end-to-end time (makespan) for executing an algorithm on GoFFish and on Giraph. This includes two key components: the time to load the data from storage, which shows the benefits of sub-graph oriented distributed storage, and the time to run the sub-graph/vertex centric computation. which shows relative benefits of the abstractions. Fig. 2(a) highlights the data loading time per graph on both platforms; this does not change across algorithms. Fig. 2(b) give the execution time as a *bar-plot* for various algorithms and datasets once data is loaded, as well as the makespan that includes the compute and load time, as a *dot-plot*. Also shown in Fig. 2(c) is the number of supersteps taken to complete the algorithm for each combination.

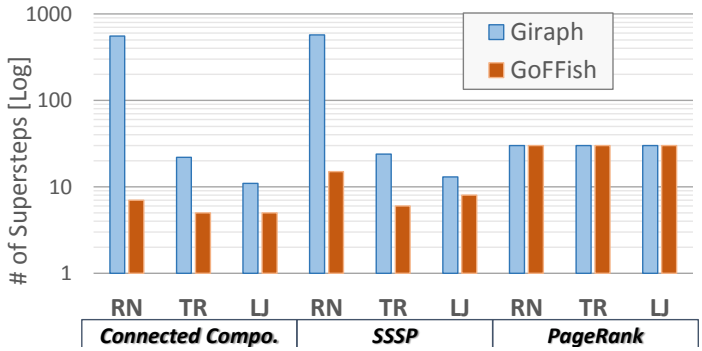
One key observation is that GoFFish's makespan is smaller than Giraph for all combinations but two, PageRank and SSSP on LJ. The performance advantage ranges from $81\times$ faster for Connected Components on RN to 11% faster for PageRank on TR. These result from abstraction, design and layout choices, as we discuss. In some, Giraph's data loading time from HDFS dominates (TR), in others, Gopher's algorithmic advantage significantly reduces the number of supersteps (RN for SSSP), while for a few, Gopher's compute time over sub-graphs dominates (PageRank on LJ).

Connected Components. Connected components identify maximally connected sub-graphs within an undirected graph such that there is path from every vertex to every other vertex in the sub-graph. The sub-graph and vertex centric algorithms are similar to



(a) Graph loading time (log scale) from disk to memory.

(b) Compute time and Makespan (incl. load) for GoFFish and Giraph for various graph algorithms and datasets (log scale)



(c) Number of supersteps (log scale) taken by Gopher and Giraph for various graph algorithms and datasets

Fig. 2: Comparison of GoFFish and Giraph for all Graph Algorithms and Datasets.

the Maximum Vertex Value algorithm [21]. In effect, we perform a *breadth first traversal* rooted at the sub-graph with the largest vertex ID, with each superstep propagating the value one level deeper till the farthest connected sub-graph is reached. Finally, all vertices are labeled with the component ID (i.e. largest vertex ID) they belong to.

The computational complexity of this algorithm is $O((d+1) \times v/p)$, where d is the diameter of the graph (specifically, of the largest connected component) constructed by treating each sub-graph as a meta vertex, v is the number of vertices in the graph and p is the number of machines (partitions). The key algorithmic optimization here comes from reducing the number of supersteps $(d+1)$ relative to the vertex centric model.

As a result, Connected Components for GoFFish performs significantly better than Giraph for all three data sets – from $1.4\times$ to $81\times$. Fig. 2(c) shows the number of supersteps is much smaller for Gopher compared to Giraph, taking between 5 (TR, LJ) and 7 (RN) supersteps for Connected Components while Giraph takes between 11 (LJ) and 554 (RN). The superstep time in itself is dominated by the synchronization overhead. The ratio of compute times improvements between Giraph and Gopher is highly correlated ($R^2 = 0.9999$) with the vertex-based diameter of the graph (Table 1), *i.e., larger the vertex-based graph diameter, greater the opportunity to reduce sub-graph-based diameter, lesser the number of supersteps, and better that Gopher algorithm performs.*

Gopher's makespan for TR graph is $21\times$ better than Giraph due to much faster data loading by GoFS ($27secs$ vs. $798secs$). Giraph's HDFS, which balances the vertices across partitions, has to move one vertex with 1M edges that takes punitively long.

Algorithm 3 Sub-Graph Centric Single Source Shortest Path

```
 1: procedure COMPUTE(SubGraph mySG, Iterator⟨Message⟩ M)
 2:     openset ← ∅                          ▶ Vertices with improved distances
 3:     if superstep = 1 then                ▶ Initialize distances
 4:         for all Vertex v in mySG.vertices do
 5:             if v = SOURCE then
 6:                 v.value ← 0              ▶ Set distance to source as 0
 7:                 openset.add(v)           ▶ Distance has improved
 8:             else
 9:                 v.value ← −∞             ▶ Not source vertex
10:     for all Message m in M do            ▶ Process input messages
11:         if mySG.vertices[m.vertex].value > m.value then
12:             mySG.vertices[m.vertex].value ← m.value
13:             openset.add(m.vertex)        ▶ Distance improved
14:             ▶ Call Dijkstras and get remote vertices to send updates
15:     remoteSet ← DIJKSTRAS(mySG, openset)
16:             ▶ Send new distances to remote sub-graphs/vertices
17:     for all ⟨remoteSG,vertex,value⟩ in remoteSet do
18:         SENDTOSUBGRAPHVERTEX(remoteSG, vertex, value)
19:     VOTETOHALT( )
```

SSSP. Intuitively, the sub-graph centric algorithm for Single Source Shortest Path (SSSP) finds the shortest distances from the source vertex to all internal vertices (i.e. not having a remote edge) in the sub-graph holding the source in one superstep using DIJKSTRAS (Alg. 3). It then sends the updated distances from the vertices having a remote edge to their neighboring sub-graphs. These sub-graphs propagate the changes internally in one superstep, and to their neighbors across supersteps, till the distances quiesce.

DIJKSTRAS has a compute complexity of $O((e \cdot log(v)))$ per superstep, where e and v are typically dominated by the largest active sub-graph. The number of supersteps is a function of the graph diameter d measured through sub-graphs, and this takes $O(d)$ supersteps. For a partitions with large number of small sub-graphs, we can exploit concurrency across c cores on that machine. While the time complexity per superstep is relatively larger for DIJKSTRAS, we may significantly reduce the number of supersteps taken for the algorithm to converge.

SSSP's compute time for GoFFish out-performs Giraph by $300\times$ and $2\times$ for RN and TR, respectively, while it is the same for LJ. Gopher takes reduced supersteps on RN and TR for SSSP, that is able to offset its higher computational complexity per superstep. But this complexity impacts LJ which has high edge density, while its small world network diameter does not reduce the number of supersteps. Hence SSSP for Gopher only matches, rather than outperforms, Giraph's compute time for LJ.

PageRank. For each superstep in a vertex centric PageRank [5], a vertex adds all input message values into sum, computes $0.15/v + 0.85 \times sum$ as its new value, and sends $value/g$ to its g neighbors. The $value$ is $1/v$ initially, for v vertices in the graph. An

equivalent sub-graph centric approach does not confer algorithmic benefits; it takes the same $\sim$30 supersteps to converge and each vertex operates independently in lock step, with an $O(30 \cdot \frac{v}{p \cdot c} \cdot g)$, for an average edge degree of g.

As shown in Fig. 2(b), Gopher under performs Giraph for PageRank for TR and LJ by $2.6\times$. It is $5.5\times$ faster for RN. TR's makespan offsets compute slowdown with data loading benefits. As observed, the fixed supersteps used for PageRank negates algorithmic benefits and the computation complexity per superstep for sub-graph centric is higher than for vertex centric. This also exacerbates the time skew across sub-graphs in a partition. For e.g., in LJ, many of the partitions complete their superstep within a range of $23 - 26secs$, but these are bound by single large sub-graphs in each partition which are stragglers, and cause 75% of the cores to be idle. Giraph, on the other hand, has uniform vertex distribution across machines and each worker takes almost the same time to complete a superstep while fully exploiting fine grained vertex level parallelism. This highlights the deficiencies of the default partitioning model used by GoFS that reduces edge cuts and balances the number of vertices per machine, *without considering the number of sub-graphs that are present per partition, and their sizes*.

6 Discussion and Conclusions

We introduce a sub-graph centric programming abstraction for large scale graph analytics on distributed systems. This model combines the scalability of vertex centric programming with the flexibility of using shared-memory algorithms at the sub-graph level. The connected nature of our sub-graphs provides stronger guarantees for such algorithms and allows us to exploit degrees of parallelism across sub-graphs in a partition. The GoFFish framework offers Gopher, a distributed execution runtime for this abstraction, co-designed with GoFS, a distributed sub-graph aware storage that pre-partitions and stores graphs for data-local execution.

The relative algorithmic benefits of using a sub-graph centric abstraction can be characterized based on the class of graph algorithm and graph. For algorithms that perform full graph traversals, like SSSP, BFS and Betweenness Centrality, we reduce the number of supersteps to a function of the diameter of the graph based on sub-graphs rather than vertices. This can offer significant reduction. However, for powerlaw graphs that start with a small vertex based diameter, these benefits are muted.

The time complexity per superstep can be larger since we often run the single-machine graph algorithm on each sub-graph. The number of vertices and edges in large sub-graph will impact this. If there are many small sub-graphs in a partition, the number of sub-graphs becomes the limiting factor as we approach a vertex centric behavior, but this also exploits multi-core parallelism. For graphs with high edge density, algorithms that are a linear (or worse) function of the number of edges can take longer supersteps.

We empirically showed that GoFFish performs significantly better than Apache Giraph. These performance gains are due to both the partitioned graph storage and sub-graph based retrieval from GoFS, and a significant reduction in the number of supersteps that helps us complete faster. This offers a high compute to communication ratio.

We do recognize some short comings, with further research opportunities. Sub-graph centric algorithms are vulnerable to imbalances in number of sub-graphs per partition and non-uniformity in their sizes. This causes stragglers. Better partitioning

to balance the sub-graphs can help. The framework is also susceptible to small-world graphs with high edge degrees that have high sub-graph level computational complexity. Our software prototype offers opportunities for design and engineering optimizations.

References

1. Gregor, D., Lumsdaine, A.: The Parallel BGL: A Generic Library for Distributed Graph Computations. In: Parallel Object-Oriented Scientific Computing (POOSC). (2005)
2. Ediger, D., Bader, D.: Investigating Graph Algorithms in the BSP Model on the Cray XMT. In: Workshop on Multithreaded Architectures and Applications (MTAAP). (2013)
3. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the gpu using cuda. In: IEEE High performance computing (HiPC). (2007)
4. Lin, J., Schatz, M.: Design patterns for efficient graph algorithms in MapReduce. In: Workshop on Mining and Learning with Graphs, ACM (2010) 78–85
5. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: ACM International Conference on the Management of Data (SIGMOD), ACM (2010) 135–146
6. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From "Think Like a Vertex" to "Think Like a Graph". Proc. of the VLDB (PVLDB) **7**(3) (2013) 193–204
7. Avery, C.: Giraph: Large-scale graph processing infrastructure on hadoop. In: Hadoop Summit. (2011)
8. Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.: Challenges in parallel graph processing. Parallel Processing Letters **17**(01) (2007) 5–20
9. Buluç, A., Madduri, K.: Parallel breadth-first search on distributed memory systems. In: IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC), ACM (2011)
10. Harshvardhan, Fidel, A., Amato, N., Rauchwerger, L.: The STAPL Parallel Graph Library. In: Languages and Compilers for Parallel Computing. Volume 7760 of LNCS. (2013) 46–60
11. Papadimitriou, S., Sun, J.: DisCo: Distributed Co-clustering with Map-Reduce. In: IEEE International Conference on Data Mining (ICDM). (2008)
12. Chen, R., Weng, X., He, B., Yang, M.: Large graph processing in the cloud. In: ACM International Conference on the Management of Data (SIGMOD), ACM (2010) 1123–1126
13. Gerbessiotis, A.V., Valiant, L.G.: Direct bulk-synchronous parallel algorithms. Journal of Parallel and Distributed Computing (JPDC) **22**(2) (1994) 251–267
14. Seo, S., Yoon, E.J., Kim, J., Jin, S., Kim, J.S., Maeng, S.: Hama: An efficient matrix computation with the mapreduce framework. In: IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE (2010)
15. Redekopp, M., Simmhan, Y., Prasanna, V.: Optimizations and analysis of bsp graph processing models on public clouds. In: IEEE Intl. Parallel & Distr. Proc. Symp. (IPDPS). (2013)
16. Salihoglu, S., Widom, J.: GPS: A Graph Processing System. In: International Conference on Scientific and Statistical Database Management (SSDBM). (2013)
17. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning and data mining in the cloud. VLDB **5**(8) (2012) 716–727
18. Shao, B., Wang, H., Li, Y.: Trinity: A distributed graph engine on a memory cloud. In: ACM International Conference on the Management of Data (SIGMOD). (2013)
19. Karypis, G., Kumar, V.: Analysis of multilevel graph partitioning. In: IEEE/ACM Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC). (1995)
20. Simmhan, Y., Kumbhare, A., Wickramachari, C.: Floe: A dynamic, continusous dataflow framework for elastic clouds. Technical report, USC (2013)
21. Kang, U., Tsourakakis, C.E., Faloutsos, C.: Pegasus: A peta-scale graph mining system implementation and observations. In: IEEE Intl. Conf. on Data Mining (ICDM). (2009)

Continuous Dataflow Update Strategies for Mission-Critical Applications

Charith Wickramaarachchi and Yogesh Simmhan
University of Southern California
Los Angeles CA 90089
{cwickram,simmhan}@usc.edu

***Abstract*—*Continuous dataflows* complement scientific workflows by allowing composition of realtime data ingest and analytics pipelines to process data streams from pervasive sensors and "always-on" scientific instruments. Such dataflows are mission-critical applications that cannot suffer downtime, need to operate consistent, and are long running, but may need to be updated to fix bug or add features. This poses the problem: *How do we update the continuous dataflow application with minimal disruption?* In this paper, we formalize different types of dataflow update models for continuous dataflow applications, and identify the qualitative and quantitative metrics to be considered when choosing an update strategy. We propose five dataflow update strategies, and analytically characterize their performance trade-offs. We validate one of these consistent, low-latency update strategies using the $\mathcal{F}\ell o\varepsilon$ dataflow engine for an eEngineering application from the Smart Power Grid domain, and show its relative performance benefits against a naïve update strategy.**

I. Introduction

eScience and eEngineering have been long contending with "Big Data", and research has led to the development of tools and middleware to gather, process, manage and analyze scientific data on cyber-infrastructure, at large scales and diversity. Scientific workflows and dataflows, in particular, have served the scientific domains well through data ingest pipelines to collect and catalog datasets generated by humans, instruments and models into repositories, as also for designing *in silico* experiments for performing scientific analysis. These efforts have predominantly focused on the *volume* and *variety* dimensions of Big Data and less so on the *velocity* of data [1].

As physical and digital sensors expand beyond small-scale deployments to pervasive presence, everywhere from our vehicles to power grids to smart phones to twitter feeds, there is a growing emphasis in harnessing these data streams for realtime ingest and analytics. Such *continuous dataflow* or stream processing is required for several reasons: the stream rates may be too fast and sustained to buffer and ingest in batch, the analysis lifetime may be shorter than the data lifetime, or for operational needs that require realtime response based on the analytics. In such cases, the dataflows operate in an *always-on* fashion and in the latter case, especially, may also need to be resilient for operations.

Cyber-Physical Systems (CPS) [1] [2] exemplify such an *eEngineering* domain where sensors and actuators deployed in physical infrastructure, such as power grids and transport networks, are monitored and controlled actively using data analytics performed on cyber-infrastructure [3], [4]. Here, continuous dataflows serve to collect and pre-process data streams reporting, say, electricity consumption events from thousands of power utility consumers [5], detect patterns in power usage [6], predict consumption demand using machine-learned models [7], and trigger notifications to increase generation or curtail demand in case of a demand-supply mismatch to avoid grid blackouts. Such continuous dataflows that perform closed-loop "observe, orient, decide, and act" (OODA) cycles [8] are also seen in transportation networks to control traffic signals based on current traffic conditions [4] and for weather prediction that actively steer CASA radars based on storm simulation models [9].

Such dataflows are mission-critical applications that cannot suffer downtime, need to operate consistently, and are also long running, often for months. The latter poses an interesting problem: *How do we update the continuous dataflow application with minimal disruption?* Such updates may be required for several reasons. We may need to fix a bug in a task in the dataflow, upgrade an analysis model that causes multiple tasks and their connectivity to change, or add/remove entire subsets of the dataflow when new types of data streams are added/removed. In fact, such *dynamism* in the application may even arise in interactive *in silico* experiments where a scientist wishes to change the continuous dataflow's composition on the fly as an experiment is running and results are emitted.

However, performing an update to continuous dataflows is deceptively non-trivial when we consider the different trade-offs in the process, both quantitative and qualitative. Performance features such as sustained output throughput of the application as an update is taking place and latency for results processed by the updated dataflow to arrive are key considerations. Time-consuming updates can also cause data loss in input streams if they cannot be buffered. Updating tasks in-place can partially mitigate these but introduce the challenge on inconsistency, where data may be processed by tasks in both the old and the updated dataflow. While this may not impact some applications, it may be a serious concern for many others and impact reproducibility. The dataflow update process is exacerbated given that they often run on distributed systems like Clouds and clusters, without shared-memory, where distributed synchronization is complex and costly.

This continuous dataflow update problem has surprisingly received inadequate attention. This can be attributed to the recent emergence of continuous and realtime analytics in eScience and eEngineering, and to the limited need for reliability of for *ad hoc* or short-duration experiments. There has been

[1] Cyber-Physical Systems Virtual Organization, http://cps-vo.org

work on check-pointing and resuming workflows [10] that can be adapted for updating dataflows but they do not address the consistency issue. These is research on provenance tracing can help detect inconsistencies between different workflows [11] or to resume incomplete workflows [12], but not applied to workflow update problems. Dataflow versioning has also been well examined [13] but tracking dataflow changes does not address enacting the actual updates. Also there has been research on dynamic reconfiguration of distributed applications [14] where they propose a design and implementation to update distributed modular applications while preserving consistently. They focus on consistency associated with state changes of the system through dynamic reconfiguration. This is related to but distinct from the consistency and performance features we propose for continuous dataflows.

In this paper, we make the following key contributions:

- We formalize different types of dataflow application updates needs (§ III), and identify the qualitative and quantitative metrics to be considered for applications when designing update strategies (§ IV).
- We introduce five different dataflow update strategies, and analytically characterize their performance metrics to help compare their trade-offs (§ V).
- We implement one of these consistent, low-latency update strategies in the $\mathcal{F}\ell o\varepsilon$ continuous dataflow engine (§ VI), and evaluate it against a simple update strategy along these metrics for our motivating application from the Los Angeles Smart Power Grid project (§ II).

II. Motivating Application

Power grids are undergoing a transformation into Smart Grids. The deployment of sensors to monitor the grid's state in realtime, such as Smart Meters at customer premises, allows electricity consumption events to be transmitted in realtime to the power utility through broadband and mesh networks. Such technology allows intelligent management of the power grid for enhanced energy efficiency and reliability, *if analyzed and acted upon in realtime* [3]. *Demand-Response optimization (DR)* is an approach to improving the grid's stability by using online predictive analytics to forecast the power demand of individual consumers, and take corrective response if predicted demand is greater than supply. As part of the DOE-sponsored *Los Angeles Smart Grid project*, the University of Southern California (USC) campus serves as a micro-grid testbed to evaluate novel forecasting models and curtailment techniques for DR before expanding them city-scale. Behavioral scientists, power systems researchers, and facility managers use data streaming from over 100 buildings and 50,000 sensors on campus that measure power usage, equipment status, ambient temperature, etc. for their experimental and operational needs.

Figure 1 shows the continuous dataflow pipeline, deployed in the campus microgrid and orchestrated by the $\mathcal{F}\ell o\varepsilon$ dataflow engine [15], to ingest and analyze events sampled from these sensors at 1∼15 min intervals [5]. The initial tasks (in blue) continuously sample different subsets of sensors, and parse, extract and validate their readings [16]. Then, these readings are semantically annotate with the building and sensor attributes, and inserted/updated them into a 4Store RDF knowledgebase (green tasks) that serves mobile apps and desktop clients at the SmartGrid Web Portal [2], which display the campus' realtime energy heatmap. Concurrently, the parsed readings are used for energy forecasting at the building level using predictive analytics models like ARIMA timeseries packaged as R scripts, regression tree models provided by Weka [7] or pattern detection using a CEP engine [17] (yellow tasks). These forecasts serve to trigger DR actions. For e.g., when the 1-hour energy use forecast for a building goes beyond a threshold, the building control center may remotely set its A/C units to a low-energy cycle, determined by a power engineer, or an automated agent may send messages, customized by a behavioral studies researcher, to its occupants to reduce their power consumption.

Over time, several such continuous dataflows will be deployed for experiments and analytics by research and operational groups. These often serve long-term studies and will require changes to the continuous dataflow's structure and logic. Such dataflow dynamism may manifest in ways that affect one or more tasks and their wiring in the dataflow. For e.g., the task that validates sensor data quality may be upgraded, a set of tasks and connections added for a new prediction model, or tasks for parsing and annotating streams may be changed as new sensor types come online (e.g. *Parse* task updated to *Parse++* task in Fig. 1). Depending on the users and their needs, these updates may need to be performed with/without loss of any messages, transient slowdown or gaps in output results from the dataflow, deterministic processing of messages, or ordering of old and new messages. For e.g., a scientist may require predictions to strictly come from sensor events processed either by the old or the updated forecast model while the fact that different versions of parser tasks were present in the dataflow may be irrelevant. A facility engineer may require no downtime when a validation task is updated while the web portal manager may accept delayed freshness of data in the semantic knowledgebase. These trade-offs determine the strategies employed to update the dataflow from one version to another, which at one extreme may be to shutdown the entire dataflow and restart an updated version, but alternatively use more nuanced techniques to perform updates to the dataflow as it is running.

III. Continuous Dataflow Update Types

We define continuous dataflow as a directed graph of *processor* (i.e. task) nodes and *channel* edges. Each processor has one input port which is the entry point of messages into the processor and one output port which is the exit point for the messages. Channels connect ports and route messages emitted from one processor to another, and determine the data dependency between them. A *continuous dataflow graph* is given as $\Gamma(\mathcal{P}, \mathcal{C})$, where the set of processors $\mathcal{P} = \{p_i \mid i \in [0, n), n = |\mathcal{P}|\}$ and the set of channels $\mathcal{C} = \{\langle p_i, p_j \rangle \mid p_i, p_j \in \mathcal{P}\}$. $\mathcal{C}$ is a *set* of directed edges; this precludes duplicate channels between the same processors in the same direction while *allowing cycles in the dataflow*.

We define *message*, μ, as a logical data unit that is consumed or produced by processors upon execution, and which flow between processors over the channels of the dataflow graph. Processors can consume and produce messages continuously in a streaming manner. The *source processors* of

[2]USC SmartGrid Portal, http://smartgrid.usc.edu

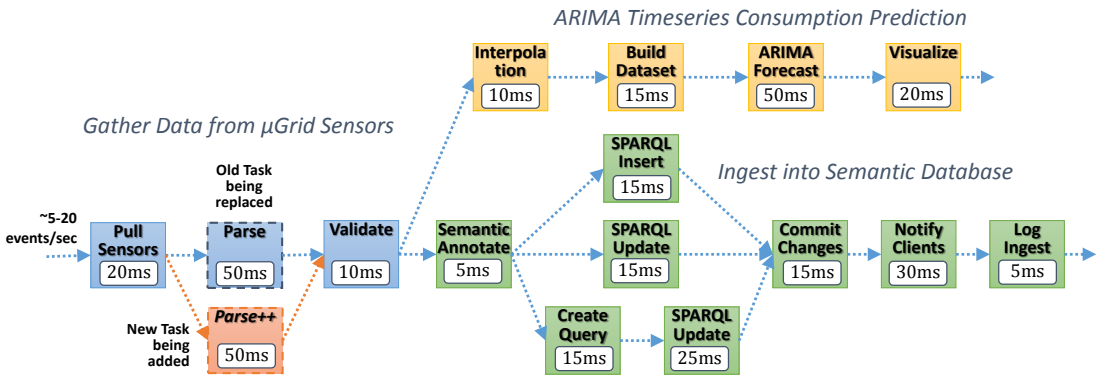


Fig. 1: Continuous Dataflow Composed in $\mathcal{F}\ell o\varepsilon$ for Realtime Sensor Data Ingest and Predictive Analytics in the USC Microgrid. Name and processing latency in milliseconds is shown for each processor. Dotted lines show channels carrying streaming messages between connected processors.

a continuous dataflow, Γ, are those whose input messages only arrive from external sources while *sink processors* are those that produce output messages only to external entities:

$$Source(\Gamma) = \{p_j \mid p_j \in \mathcal{P} \text{ and } \nexists c_k = \langle p_i, p_j \rangle \forall p_i \in \mathcal{P}, c_k \in \mathcal{C}\}$$

$$Sink(\Gamma) = \{p_i \mid p_i \in \mathcal{P} \text{ and } \nexists c_k = \langle p_i, p_j \rangle \forall p_j \in \mathcal{P}, c_k \in \mathcal{C}\}$$

The *arrival channel* $\Xi_p(\mu) \in \mathcal{C}$ of a message μ for a processor p is defined as the channel on which that message arrives at p to be consumed by it. We define $\Pi(\mu) \in \mathcal{P}$ as the *parent processor* μ was produced by. $\mathcal{M}(\mu)$ is the *parent-set of messages* consumed by the parent processor to generate μ:

$$\mathcal{M}(\mu) = \begin{cases} \emptyset, \text{ if } \Pi(\mu) \in Source(\Gamma) \\ \textit{messages consumed by } \Pi(\mu) \textit{ to emit } \mu, \text{otherwise} \end{cases}$$

Given the above formalism, we extend it to define the concept of *dataflow versioning*. Let $T = \{\tau_0, \tau_1, \tau_2, \dots\}$ be the set of times at which updates to the dataflow are initiated. We assume that a new update is not started until the previous update has been completed. Update Time (UT) is the time between starting an update and the updated dataflow version starting to process messages. So, $\tau_{i+1} - \tau_i \geq UT$.

For a dataflow $\Gamma(\mathcal{P}, \mathcal{C})$, we define its *versioned dataflow* during a time t, where $\tau_s < t < \tau_{s+1}$, as $\Gamma^s(\mathcal{P}^s, \mathcal{C}^s)$. Here, $\mathcal{P}^s$ and $\mathcal{C}^s$ are the set of versioned processors and channels present in the dataflow after the update at time τ_s and before the next update at τ_{s+1}. A processor $p_i \in \mathcal{P}^s$ has an associated *version* given by $\mathcal{V}^s(p_i)$. If the application logic of $p_i \in \mathcal{P}^s$ and $p_i \in \mathcal{P}^{s+1}$ differ, then $\mathcal{V}^s(p_i) \neq \mathcal{V}^{s+1}(p_i)$. and also if $p_i \in \mathcal{P}^s \bigcap \mathcal{P}^{s+1}$ then $\mathcal{V}^s(p_i) = \mathcal{V}^{s+1}(p_i)$.

We identify four **types of dataflow updates** that are performed on continuous dataflow graphs based on how the dataflow structure changes from one version to another. These vary from just one or more processors changing without any changes to the channel connectivity, to entire subgraphs in the dataflow graph being replaced by another. These update types are not meant to be exhaustive, but rather to capture the common types of runtime updates on continuous dataflows.

Consider a dataflow update that changes a versioned dataflow from $\Gamma^s(\mathcal{P}^s, \mathcal{C}^s)$ to $\Gamma^{s+1}(\mathcal{P}^{s+1}, \mathcal{C}^{s+1})$.

1) **Processor Update Type:** A dataflow update is of a processor update type if only the version of one or more of its processors change without any change to the number of processors or the channel connectivity. Formally, $\Gamma^s \rightarrow \Gamma^{s+1}$ is a processor update type iff:

$$\iff \begin{cases} |\mathcal{P}^s| = |\mathcal{P}^{s+1}| \ \& \\ |\mathcal{C}^s| = |\mathcal{C}^{s+1}| \ \& \\ \forall c_k = \langle p_i, p_j \rangle \in \mathcal{C}^{s+1} \exists c_l = \langle p_i, p_j \rangle \in \mathcal{C}^s \ \& \\ \exists p_i \in \mathcal{P}^{s+1}, p_i \in \mathcal{P}^s : \mathcal{V}^s(p_i) \neq \mathcal{V}^{s+1}(p_i) \end{cases}$$

2) **Channel Update Type:** When a dataflow update causes only the number of channels or their connectivity to change without the number of processors or their versions to change, we call this a channel update type. Formally, $\Gamma^s \rightarrow \Gamma^{s+1}$ is a channel update type iff:

$$\iff \begin{cases} |\mathcal{P}^s| = |\mathcal{P}^{s+1}| \ \& \\ \forall p_l \in \mathcal{P}^s \exists p_m \in \mathcal{P}^{s+1} : l = m, \mathcal{V}^s(p_l) = \mathcal{V}^{s+1}(p_m) \ \& \\ \begin{cases} |\mathcal{C}^s| \neq |\mathcal{C}^{s+1}| \text{ or} \\ \mathcal{C}^s \cap \mathcal{C}^{s+1} \neq \mathcal{C}^s \end{cases} \end{cases}$$

3) **Independent Subgraph Update Type:** When an update causes one or more processor versions to be changed without adding or removing processors, *and* causes the number of channels and/or their connectivity to change, we term this an independent subgraph update. This is akin to having both processor and channel updates taking place; the independent refers to the fact that the changes are not restricted to one part of the dataflow graph. Formally, $\Gamma^s \rightarrow \Gamma^{s+1}$ is an independent subgraph update type iff:

$$\iff \begin{cases} |\mathcal{P}^s| = |\mathcal{P}^{s+1}| \ \& \\ \exists p_i \in \mathcal{P}^{s+1}, p_i \in \mathcal{P}^s : \mathcal{V}^s(p_i) \neq \mathcal{V}^{s+1}(p_i) \ \& \\ \begin{cases} |\mathcal{C}^s| \neq |\mathcal{C}^{s+1}| \text{ or} \\ \mathcal{C}^s \cap \mathcal{C}^{s+1} \neq \mathcal{C}^s \end{cases} \end{cases}$$

4) **Connected Subgraph Update Type:** When an update causes a subgraph in the dataflow graph to be replaced by another subgraph, we call this a connected subgraph update. Here, the updates are localized to one section of the dataflow and causes processors and channels in this section to be added/updated/removed. So the two dataflows share a common

connected subgraph but also have a connected subgraph that appear in each independently. Formally, $\Gamma^s \rightarrow \Gamma^{s+1}$ is a connected subgraph update type iff:

$$\iff \begin{cases} (P^s \bigcap P^{s+i}, C^s \bigcap C^{s+i}) \text{ forms a connected graph } \& \\ (P^{s+1} - P^s, C^{s+1} - C^s) \text{ forms a connected graph} \end{cases}$$

We define the term *message dependency trace* for a message μ_0 that has been generated by its parent processor $\Pi(\mu_0)$ as $\Delta(\mu_0) = (\mathbb{P}, \mathbb{C})$ where $\mathbb{P}$ and $\mathbb{C}$ are the set of causally-dependent processors and channels that caused this message to be produced. Intuitively, this trace is the *provenance* of μ_0. We define $\mathbb{P}$ and $\mathbb{C}$ recursively using the following rules:

Basis: $\Pi(\mu_0) \in \mathbb{P}$
Induction: $\Pi(\mu_x) \in \mathbb{P} \Rightarrow \Pi(\mu_{x'}) \in \mathbb{P}, \forall \mu_{x'} \in \mathcal{M}(\mu_x)$

Basis: $\Xi_{p_i}(\mu') \in \mathbb{C}$, $\forall \mu' \in \mathcal{M}(\mu_0)$
Induction: $\Xi_{p_i}(\mu_x) \in \mathbb{C} \Rightarrow \Xi_{p_i}(\mu_{x'}) \in \mathbb{C}, \forall \mu_{x'} \in \mathcal{M}(\mu_x)$

This definition follows even when cycles are present in the dataflow. Since $\mathbb{P}$ and $\mathbb{C}$ are sets, the number of elements in the trace is independent of number of iterations a message has participated in and, effectively, rolls in the iterations to one.

IV. Qualitative and Quantitative Update Metrics

Users need to consider different performance guarantees when updating continuous dataflow applications. Update strategies offer various trade-offs when changing the dataflow from one version to another. Here, we identify and formalize these metrics to help characterize the strategy.

1) **Consistency.** In-place update strategies may modify the dataflow as it continues to process messages. When an update consists of changes to more then one processor/channel, then an update strategy may cause messages processed by an older processor/channel version to also be processed by the newer version. The reverse may also be true when dataflow cycles are present. This can cause inconsistent results since messages have been processed through an intermediate dataflow structure that is neither of the two versions. We call a message as consistently processed if it and it's causal messages and processors were processed only by processors and channels belonging to a single dataflow version.
Formally, if message μ is emitted by a sink processor of a dataflow being updated from $\Gamma^s \rightarrow \Gamma^{s+1}$, and the message trace of μ is $\Delta(\mu) = (\mathbb{P}, \mathbb{C})$, then μ is a consistently processed message iff $\mathbb{P} \subset \mathcal{P}^s$ and $\mathbb{C} \subset \mathcal{C}^s$, or $\mathbb{P} \subset \mathcal{P}^{s+1}$ and $\mathbb{C} \subset \mathcal{C}^{s+1}$. In case of the former, μ is considered to have been consistently processed by dataflow version Γ^s, and in the latter case, by dataflow version Γ^{s+1}. An update strategy is consistent if all messages emitted between times τ_s to τ_{s+1} are consistent.

2) **Interleaved & Delineated.** In-place updates can causes messages to be present in two version of the dataflow at the same time, and follow different paths. Hence the sink processor of the dataflow may emit "faster" messages from the new version before it emits "slower" messages from the old version. In such cases, termed interleaved, it is not possible to have a strictly time boundary that separates results from the old and new dataflows. Formally, for a dataflow update from $\Gamma^s \rightarrow \Gamma^{s+1}$, let t_l be the time when the last message is processed and emitted by Γ^s and t_f the time when the first message is processed and emitted by Γ_{s+1}. An update strategy is delineated if $t_f > t_l$, and it is interleaved otherwise.

3) **Refresh Latency.** The time between when the update was initiated and messages processed by the new dataflow version emerges determines the freshness of the dataflow results. For a dataflow update from $\Gamma^s \rightarrow \Gamma_{s+1}$ initiated at τ_{s+1}, let t_f be the time when the first message μ emitted by the dataflow with message trace $\Delta(\mu) = (\mathbb{P}, \mathbb{C})$ such that $\mathbb{P} \bigcap \mathcal{P}^{s+1} \neq \emptyset$ or $\mathbb{C} \bigcap \mathcal{C}^{s+1} \neq \emptyset$. Then the refresh latency is given by $(t_f - \tau_{s+1})$.

4) **Lag Latency.** We define lag latency as the time duration between when an update was started and the time at which the last message is emitted from the old version of the dataflow. This gives an indication of the delay in flushing messages from the old dataflow. For a dataflow update from $\Gamma^s \rightarrow \Gamma_{s+1}$, let t_l be the time at which the last message μ is emitted with message trace $\Delta(\mu) = (\mathbb{P}, \mathbb{C})$ such that $\mathbb{P} \bigcap \mathcal{P}^s \neq \emptyset$ or $\mathbb{C} \bigcap \mathcal{C}^s \neq \emptyset$. Then the lag latency is given by $(t_l - \tau_{s+1})$.

5) **Throughput.** Dataflow updates can have a disruptive impact on the output throughput observed from the dataflow as the update is taking place. For offline updates, this throughput can briefly drop to zero, and it can take some time before the throughput rate recovers as the input messages buffered during the update are eventually processed. Some applications may require strict bounds on the loss in throughput as seen at the sink processors. We recognize the output throughput of the dataflow as a key metric in evaluating the update strategies.

6) **Message Loss.** Some update strategies may force the dataflow to drop messages by shutting it down immediately rather than gracefully. This can help improve the refresh latency or ensure message consistency. We compare the update strategies based on whether they cause in-flight messages to be lost as this may impact mission-critical operations. Separately, we assume that input messages can be buffered before they enter the source processors. So the message loss feature is restricted to those that have entered the dataflow.

V. Dataflow Update Strategies and Analysis

In this section, we introduce several update strategies of increasing complexity and offering different trade-offs based on the metrics. We discuss the algorithm for the strategy and offer analytical estimates for their metrics for the different update types based on the formalism introduced earlier.

We define several terms to help quantify the performance metrics for the strategies. *Message Wave* (MV_{p_i,p_j}) between two processors p_i and p_j denotes the set of messages that are generated by p_j as result of one message that is consumed by p_i. Hence, the message dependency traces of all messages in MV_{p_i,p_j} will contain the both processors p_i and p_j, and the messages will be causal dependent on the same message that was consumed by p_i. *Wave Head time* ($W^{head}_{p_i,p_j}$) and *Wave Tail time* ($W^{tail}_{p_i,p_j}$) identify the time it takes for the first and last message in a wave to pass between the two processors. Let MV_{p_i,p_j} be the message wave for message μ_0 consumed by p_i, and the function $time_{p_i}(\mu)$ denote the time at which message μ was consumed by processor p_i. Then:

$$W^{head}_{p_i,p_j} = \text{MIN}\{time_{p_j}(\mu_k) \mid \forall \mu_k \in MV_{p_i,p_j}\} - time_{p_i}(\mu_0)$$

$$W^{tail}_{p_i,p_j} = \text{MAX}\{time(\mu_k) \mid \forall \mu_k \in MV_{p_i,p_j}\} - time_{p_i}(\mu_0)$$

We define *selectivity* of a processor p_i as $\sigma(p_i) = m : n$, the ratio between the number of output messages n generated for m input messages consumed by the processor, where $m, n > 0$. Given the selectivity of a dataflow's processors, the latency for a processor to execute and the input message rate, we can mathematically bound $W^{head}_{p_i,p_j}$ and $W^{tail}_{p_i,p_j}$ by calculating the shortest and longest paths in the dataflow graph between the processors. We omit details of this and assume the wave head and tail times are available. The *Deployment time* (DF_Γ) is defined as the time taken to deploy processors and instantiate a dataflow Γ, while *Input message rate* (I_{p_i}) for a processor p_i is the aggregated rate of messages arriving at its input channels.

A. Naïve Consistent Lossy Update (NCL)

Here, when an update request is received, the input streams are paused and the existing dataflow is terminated immediately. All messages that are being processed by the dataflow are lost. The new dataflow Γ^{s+1} is deployed immediately from scratch and the input message streams resumed. This technique can be applied to all the update types in § III.

- Since the dataflow is stopped upon the update request and subsequent messages are accepted only after the new dataflow is deployed, this update is *consistent*.
- Messages are *delineated* since the dataflow is stopped immediately upon the update request.
- *Refresh Latency* $\leq DT_{\Gamma^{s+1}} + \text{MIN}\{W^{head}_{p_1,p_2} \mid \forall p_1 \in Source(P^{s+1}), p_2 \in Sink(P^{s+1})\}$.
- *Lag Latency* $= 0$, since the dataflow is stopped immediately upon the update request.
- *Throughput* $= 0$ at time τ_{s+1} for a duration of $DT + \text{MIN}\{W^{head}_{q,r} \mid \forall q \in Source(\mathcal{P}^{s+1}), r \in Sink(\mathcal{P}^{s+1})\}$
- This is a *lossy* update strategy, and all messages that are in-flight at time τ_{s+1} are lost. The exact bound for the number of lost messages can be calculated from the selectivity of processors and input data data rate, but is omitted for brevity

B. Naïve Consistent High-latency Update (NCH)

This is a variant of the above update that pauses the input stream and flushes in-flight messages to completion before terminating the dataflow and deploying the new version. A time to live (TTL) can bound the time required to flush messages, given by $TTL_{\Gamma^s} = \text{MAX}\{W^{tail}_{p_1,p_2} \mid \forall p_1 \in Source(\mathcal{P}^s), p_2 \in Sink(\mathcal{P}^s)\}$. This update strategy guarantees consistency and delineation while increasing the refresh time, the lag time, and prolonging the duration of low throughput.

- This update is *consistent*
- This update is *delineated*
- *Refresh Latency* $\leq TTL_{\Gamma^s} + DT_{\Gamma^{s+1}} + \text{MIN}\{W^{head}_{p_1,p_2} \mid \forall p_1 \in Source(\mathcal{P}^{s+1}), p_2 \in Sink(\mathcal{P}^{s+1})\}$
- *Lag Latency* (upper bound) $= TTL_{\Gamma^s}$
- *Throughput* $= 0$ starting at time $\tau_{s+1} + TTL_{\Gamma^s}$ and for a duration of $\text{MIN}\{W^{head}_{p_1,p_2} \mid \forall p_1 \in Source(\mathcal{P}^{s+1}), p_2 \in Sink(\mathcal{P}^{s+1})\}$
- There is *no message loss*

C. High-Throughput Inconsistent Update (HTI)

This update model, while not providing message consistency, ensures a *loss-free* update with *∼no impact on the throughput*. The algorithms differ for the update types, but broadly perform in-place updates aggressively upon update request while retaining in-flight messages and resuming them on the new dataflow, causing *inconsistency* and *interleaving*.

Processor Update Type: New processors versions are instantiated independently and concurrently upon update request while retaining the old versions. Messages actively being executed by an old processor are completed while subsequent messages on its input channel are executed by the new version of the processor. The old processor will be discarded once it finishes processing its active message(s). Let DT_{p_i} be the time taken to deploy and instantiate a new processor p_i.

- *Refresh latency* $\leq \text{MIN}\{DT_{p_1} + 1/I_{p_1} + W^{head}_{p_1,p_2} \mid \forall p_1 \in (\mathcal{P}^{s+1} - \mathcal{P}^s), p_2 \in Sink(\mathcal{P}^{s+1})\}$
- *Lag Latency* $\leq \text{MAX}\{W^{tail}_{p_1,p_2} \mid \forall p_1 \in (\mathcal{P}^s - \mathcal{P}^{s+1}), p_2 \in Sink(\mathcal{P}^s)\}$

Channel Update Type: Channels are updated independently in parallel upon an update request while retaining old channels. Messages passing through old channels are routed as before while new messages subsequently produced by the processors are sent on the new channels. Old channels are discarded once all in-flight messages over the old channels are transmitted to their destination. Let DT_c be the time taken to deploy and connect a new channel c.

- *Refresh Latency* $\leq \text{MIN}\{DT_c + 1/I_{p_1} + W^{head}_{p_1,p_2} \mid \forall c \in (\mathcal{C}^{s+1} - \mathcal{C}^s), p_1 \in \mathcal{P}^{s+1}, p_2 \in Sink(\mathcal{P}^{s+1})$ where $c = (p_1, p_2)\}$
- *Lag Latency* $\leq \text{MAX}\{W^{tail}_{p_1,p_2} \mid \forall c \in (\mathcal{C}^s - \mathcal{C}^{s+1}), p_1 \in \mathcal{P}^{s+1}, p_2 \in Sink(\mathcal{P}^{s+1}), c = (p_1, p_2)\}$

Independent Subgraph Update Type: This combines the above two approaches and updates both processors and channels concurrently upon the update request. The metrics are derived as above; we omit further discussion for brevity.

Connected Subgraph Update: When an update request is received, the new subgraph with all its processors and channels is deployed and connected to main dataflow while retaining the old subgraph being replaced. Processors from the main dataflow that connect to the new subgraph will emit new messages to it while messages present in the old subgraph will proceed to completion. The old subgraph will be discarded once there are no messages in its components. Let DT_{sg} be the time to deploy all processors and channels in the new subgraph.

- *Refresh Latency* $\leq DT_{sg} + \text{MIN}\{\ \text{MAX}\{1/I_{p_1} + W^{head}_{p_1,p_2} \mid \forall p_1 \in (\mathcal{P}^{s+1} - \mathcal{P}^s), p_2 \in Sink(\mathcal{P}^{s+1})\},\ \text{MAX}\{1/I_{p_1} + W^{head}_{p_1,p_3} \mid \forall c \in (\mathcal{C}^{s+1} - \mathcal{C}^s), p_1 \in (\mathcal{P}^{s+1} - \mathcal{P}^s), p_3 \in Sink(\mathcal{P}^{s+1})$ where $c = (p_1, p_3)\}\}$
- *Lag Latency* $\leq \text{MAX}\{\ \text{MAX}\{W^{tail}_{p_1,p_2} \mid \forall p_1 \in (\mathcal{P}^s - \mathcal{P}^{s+1}), p_2 \in Sink(\mathcal{P}^s)\},\ \text{MAX}\{W^{tail}_{p_2,p_3} \mid \forall c \in (\mathcal{C}^{s+1} - \mathcal{C}^s), p_3 \in Sink(\mathcal{P}^{s+1})$ where $c = (p_1, p_2)\}\}$

D. Message-Versioned Consistent Update (MVC)

This strategy offers *consistent* and *loss-less* update with *low impact on throughput*. Intuitively, this model tags messages with a *version property* that is set to the dataflow version s for Γ^s when messages enter the source processors. Under normal non-update conditions, child messages inherit the same version

as the parent message set. However, after an update request the rules discussed below define how this property is propagated and used to perform a consistent, though *interleaved*, update.

Processor Update Type: When an update request is received at time τ_{s+1}, new processors are deployed and instantiated in parallel while old processors continue to run. The source processors are notified of the new version $s+1$ after which new input messages entering the dataflow will have their version property as $s+1$, coexisting with in-flight messages having version s. When a message with version s or $s+1$ arrives at a processor that is retained in the new dataflow version, it will generate messages with the same version. If a message arrives at a processor having both old and new processors, then the one corresponding to the message's version will be chosen and generated messages will retain the source message's version. The presence of cycles can cause old processors to live in perpetuity. We discard an old processor p_i after its Time To Live (TTL_p) period expires, given by $\text{MAX}\{W^{tail}_{p_1,p_i} \mid \forall p_1 \in Source(\Gamma^s)\} + n \times W^{tail}_{p_i,p_i}$. Here, n is the maximum number of iterations that a message wave can pass through the old processor if it had cycles (i.e., $W^{tail}_{p_i,p_i} > 0$. Let DT_p be the time to deploy and instantiate all new processors and notify the source processor(s) of the new version.

- *Refresh Latency* $\leq DT_p + \text{MIN}\{1/I_{p_1} + W^{head}_{p_1,p_2} \mid \forall p_1 \in Source(\Gamma^{s+1}), p_2 \in Sink(\Gamma^{s+1})\}$
- *Lag Latency* $\leq DT_p + \text{MAX}\{W^{tail}_{p_1,p_2} \mid \forall p_1 \in Source(\Gamma^s), p_2 \in Sink(\Gamma^s)\}$

Channel Update Type: The channel update is similar to the processor update as in the new channels are updated in parallel while retaining the old ones, and the source processors are notified of the new version $s+1$ to tag incoming messages. The decision on which channel, old or new, to send messages on is taken by each processor *after* it has generated an output message and depends on the version property of the message. The TTL_c for old channel $c = p_i, p_j$ is given by $\text{MAX}\{W^{tail}_{p_1,p_j} \mid \forall p_1 \in Source(\Gamma^s)\} + (n * W^{tail}_{p_i,p_i})$, where n is the maximum iterations a message wave will loop through p_i. Let DT_c be the time to deploy and connect all new channels and notify the source processor(s) of the new version.

- *Refresh Latency* $\leq DT_c + \text{MIN}\{1/I_{p_1} + W^{head}_{p_1,p_2} \mid \forall p_1 \in Source(\Gamma^{s+1}), p_2 \in Sink(\Gamma^{s+1})\}$
- *Lag Latency* $\leq DP_c + \text{MAX}\{W^{tail}_{p_1,p_2} \mid \forall p_1 \in Source(\Gamma^s), p_2 \in Sink(\Gamma^s)\}$)

Independent Subgraph Update Type: This update type combines the techniques of both processor and channel update types, and the latencies and TTL_{sg} can be derived similar to above. Further discussion is omitted for brevity.

Connected Subgraph Update Type: One can treat the connected subgraph as a specialization of the processor update where the entire subgraph behaves like a single virtual processor. Hence, as for the processor update, when an update request is received, the new subgraph is deployed and connected to the main dataflow while the old subgraph continue to run. The source processors are notified of the new version causing future messages entering the dataflow to be tagged with version $s+1$. Messages with version $s+1$ are routed to the new subgraph while those with version s go to the old subgraph. Existing messages with version s run to completion, and the old subgraph is discarded after a TTL_{sg} given as the $\text{MAX}\{TTL_p, TTL_c\}$, where TTL_p and TTL_c are from the processor and channel update types. Let DT_{sg} be the time to deploy and connect all components of the new subgraph and notify the source processor(s) of the new version.

- *Refresh Latency* $\leq DT_{sg} + \text{MIN}\{1/I_x + W^{head}_{p_1,p_2} \mid \forall p_1 \in Source(\Gamma^{s+1}), p_2 \in Sink(\Gamma^{s+1})\}$
- *Lag Latency* $\leq DT_{sg} + \text{MAX}\{W^{tail}_{p_1,p_2} \mid \forall p_1 \in Source(\Gamma^s), p_2 \in Sink(\Gamma^s)\}$

E. Path-Versioned Consistent Update (PVC)

One of the short-comings of the previous update strategies was that the shortest refresh latency was bounded by the time it took for a new message to pass from the source to the sink processors. However, when updates do not affect large portions of the dataflow graph, one can intuitively consider "upgrading" in-flight messages to the new version provided they have only been processed by processors that are present in both versions of the dataflow. In other words, if the causal dependency trace for a message in-flight in the old dataflow version is identical to a similar message in the new dataflow version, then the message from the old dataflow can be considered to have been processed by the new dataflow. This approach allows us to perform "back in time" updates and is advantageous when processors or channels toward the end of the dataflow is updated since more in-flight messages satisfy the condition.

This strategy offers *consistent, loss-free, interleaved* updates with *∼no throughput reduction* and provide refresh and lag latencies that are shorter than even the MVC strategy. Here, we utilize the *message dependency trace* which is carried as a *trace property* by each message. The message dependency trace definition provides recursive induction clauses to set its value. Each processor and channel follows these clauses to update the trace property for messages that pass through them.

As before, upon receipt of the update request, the relevant processors and/or channels will be deployed and instantiated independently while the old processors/channels are retained. The distinction comes in the message routing where the entire trace is examined. The following algorithm gives the steps taken by a processor in routing its input message to the old or new version. A similar approach can be used for routing output messages to old/new channels.

```
function RouteInMessage(msg, oldProc, newProc)
    (ℙ, ℂ) ← msg.TraceProperty()
    isNewVersion ← true
    for all p_i ← ℙ do
        if NOT (p_i ∈ 𝒫^s ⋂ 𝒫^{s+1} OR ℙ ⋂ 𝒫^{s+1} ≠ ∅) then
            isNewVersion ← false
            break
        end if
    end for
    if isNewVersion then
        newProc.Invoke(msg)
    else
        oldProc.Invoke(msg)
    end if
end function
```

We discuss metrics for the **Processor Update Type** and leave an analysis of other update type for future work due

to lack of space. The TTL_p after which old processors can be discarded is given by $\text{MAX}\{W^{tail}_{p_i,p_j} + n \times W^{tail}_{p_j,p_j} \mid p_i \in \mathcal{P}^{s+1}$, all predecessors of $p_i \in (\mathcal{P}^s - \mathcal{P}^{s+1})\}$ where n is the maximum number of iterations of a message wave through p_j.

- *Refresh latency* $\leq DT_p + \text{MIN}\{1/I_{p_1} + W^{head}_{p_1,p_2} \mid x \in \mathcal{P}^{s+1}$, all predecessors of $p_1 \in (\mathcal{P}^s - \mathcal{P}^{s+1}), p_2 \in Sink(\Gamma^{s+1})\}$
- *Lag Latency* $\leq DT_p + \text{MAX}\{W^{tail}_{p_1,p_2} \mid p_1 \in \mathcal{P}^{s+1}$, all predecessors of $p_1 \in (\mathcal{P}^s - \mathcal{P}^{s+1}), p_2 \in Sink(\Gamma^{s+1})\}$

VI. Implementation & Results

We implemented Message-Versioned Consistent (MVC) update strategy for the processor update type within our *Floe* [15] continuous dataflow engine that is used within the USC Smartgrid Project for the orchestrating the dataflow shown in Fig. 1. MVC offers a suitable balance between complexity and performance. We incorporate a message context header within each message that flows between the tasks that is updated with the version property according to the inheritance rules for the MVC. Later, this context can also encapsulate the trace property for PVC update.

We compare the MVC strategy against the Naïve Consistency Lossy (NCL) update strategy using the dataflow from Fig. 1 where the task "Parse" is updated to "Parse++". We ran the experiment in a pseudo-distributed environment where all the processors are deployed within independent containers in the same machine. Our experiments use input message rates of 5 msg/sec and 10 msg/sec, which give an effective output throughput of 15 msg/sec and 30 msg/sec. Fig. 2a compares the expected and observed refresh and lag latencies based on our metric analysis and the experiments. These are plotted for the two update strategies and two rates. We see that our analysis produces latency values that are close to the observed values. The deviation can be attributed to transport overheads not factored in the analysis and non-uniform performance of the tasks. We also see that MVC offers refresh latencies that are orders of magnitude faster than the NCL strategy. However, NCL does have zero lag latency due to its immediate termination of the dataflow which also causes message losses.

We also analyzed the output throughput of the dataflows, as measured at the "Log Ingest" task for the update strategies at 5 and 10 msg/sec input rates. Fig. 2b and 2c show that for both strategies, the throughput is steady at 15 and 30 msg/sec before the update is done. But the NCL's throughput drops to zero for $\sim$15 secs as the dataflow is terminated, the new version deployed and the messages start arriving at the output, eventually reaching a steady rate. On the other hand, we do not even see a blip in the throughput for the MVC update. This minimal impact of throughput is barely noticeable to the end user even as consistent messages processed by the new dataflow version is generated. We also manually verified that the experimental results confirmed the expected consistency and interleaved/delineated metrics for the update strategies.

VII. Related work

Scientific workflow systems [18], [19] are popular for modeling and composing eScience applications. These applications span from operational workflows that manage data ingest [20], [18] to exploratory ones used by scientists for *in silico* experiments [19], [21]. Scientific workflows are often composed as a directed acylic graphs (DAGs) with data processing units connected by control or data flow edges. Typically, changes to a workflow are primarily done through parameterization of a task so that passing different inputs to it causes different logic to execute, like a switch statement [22]. But this is not adequate for being able to update the logic of the tasks and dataflow structure at runtime. Steam processing systems [23], [24], [25] complement workflow systems by operating on realtime data. Online stream update problems are as less studied. There is some work on stream migration in a distributed system[26] that considers some relevant issues but not in the context of task or channel updates.

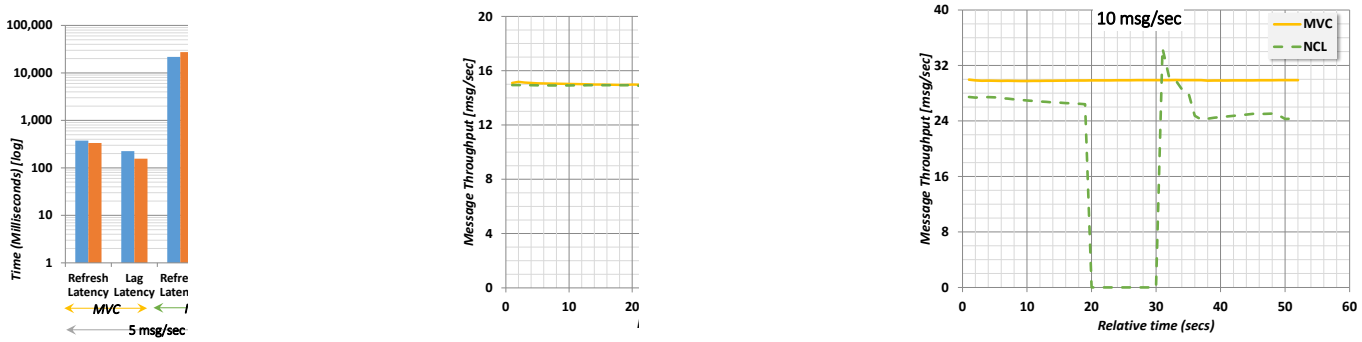
Enterprise service bus (ESB) [27] are yet another compositional pattern used in service-oriented architectures. ESB act as an intermediate broker that mediates message between loosely connected services. Updates to the ESB configuration are done atomically, corresponding to changing just one channel, rather than the different update models we discuss. While ESB atomic update techniques are useful in a centralized system, we consider a distributed update problem where central coordination is minimized to improve performance and the processors update independently, using the message context for capturing light-weight states.

Work on check-pointing and resuming of dataflows [10] can assist naïve update models to minimize message losses. Research on provenance tracing in dataflows [11] can help detect inconsistencies of data after updates are performed. Our own prior work married these two notions of provenance and dataflow reliability to provide reliability and consistency for batch workflow pipelines used for periodically loading astronomy data [20], through it did not offer analytical bounds. There the focus was on fault-tolerance on commodity clusters. While these techniques provide the tools for enhancing the update strategies, our contribution in this paper delves into the conceptual, analytical and empirical space of the continuous dataflow update problem.

More broadly, runtime reconfiguration techniques for distributed applications [14] have considered similar issues. There, production systems with distributed intercommunicating components expect high availability and consistency guarantees. Dynamic reconfiguration focuses on being able to update the distributed components while preserving their state consistently. In our related problem, we need to consider the consistent state of the message that are streaming continuously rather the state of the distributed tasks. Further, performance metrics such as message loss and latency are key considerations we examine.

VIII. Conclusion

In this paper, we have recognized the value of online updates to continuous dataflows to support long running science and operational applications in eEngineering, and the gap in existing literature. We have formalized several update models along with metrics to evaluate update strategies to allow users to make informed trade-offs. We have proposed two naïve and three advanced update strategies that offer



(a) Refresh (b) Observed Thr (c) Observed Throughput (10 msg/sec input)

Fig. 2: Experimental and/or Analytical latency and throughput values when performing Message-Versioned Consistent Update (MVC) and Naïive Consistent Lossy Update (NCL) on USC Microgrid dataflow with input rates of 5, 10 msg/sec.

different characteristics, and analyzed their metrics to bound the performance. Two of these, MVC and NCL, are validated and comparatively evaluated for the Smart Grid domain using real dataflows. They bear out our analysis and also show the performance benefits of MVC.

There is much to be explored in this space, particularly on introducing additional scoped constrains for performing modular and incremental updates, and also on what this means for dataflow verifiability. Our PVC strategy uses message traces that are akin to data provenance, and this offers some insights into possible solutions. Implementing and evaluating this strategy, as also others in a distributed environment, is part of our future work.

REFERENCES

[1] gartner, "Big Data," http://www.gartner.com/it-glossary/big-data/, [Online; accessed 19-5-2013].

[2] H. K. Manfred Broy and R. E. Achatz, "Agenda cyber physical systems recommendations for the 8th framework programme." Acatech, 2010.

[3] S. D. Ramchurn, P. Vytelingum, A. Rogers, and N. R. Jennings, "Putting the'smarts' into the smart grid: a grand challenge for artificial intelligence," *Communications of the ACM*, vol. 55, no. 4, 2012.

[4] J. Dunkel, A. Fernández, R. Ortiz, and S. Ossowski, "Event-driven architecture for decision support in traffic management systems," *Expert Systems with Applications*, vol. 38, no. 6, 2011.

[5] Y. Simmhan, S. Aman, A. Kumbhare, R. Liu, S. Stevens, Q. Zhou, and V. K. Prasanna, "Cloud-based software platform for data-driven smart grid management," in *CiSE*, 2013.

[6] Q. Zhou, Y. Simmhan, and V. Prasanna, "Incorporating semantic knowledge into dynamic data processing for smart power grids," in *ISWC*, ser. Lecture Notes in Computer Science, 2012, vol. 7650.

[7] S. Aman, Y. Simmhan, and V. K. Prasanna, "Improving energy use forecast for campus micro-grids using indirect indicators," in *DDDM*, 2011.

[8] Z. Liu, D.-s. Yang, D. Wen, W.-m. Zhang, and W. Mao, "Cyber-physical-social systems for command and control," *Intelligent Systems, IEEE*, vol. 26, no. 4, 2011.

[9] B. Plale, D. Gannon, D. Reed, S. Graves, K. Droegemeier, B. Wilhelmson, and M. Ramamurthy, "Towards dynamically adaptive weather analysis and forecasting in lead," in *ICCS*. Springer, 2005.

[10] G. Kandaswamy, A. Mandal, and D. A. Reed, "Fault tolerance and recovery of scientific workflows on computational grids," in *CCGRID*, 2008.

[11] Z. Bao, S. Cohen-Boulakia, S. B. Davidson, and P. Girard, "Pdiffview: viewing the difference in provenance of workflow results," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, 2009.

[12] I. Foster, J. Vockler, M. Wilde, and Y. Zhao, "Chimera: A virtual data system for representing, querying, and automating data derivation," in *SSDBM*. IEEE, 2002, pp. 37–46.

[13] T. Ellkvist, D. Koop, E. W. Anderson, J. Freire, and C. Silva, "Provenance and annotation of data and processes," 2008, ch. Using Provenance to Support Real-Time Collaborative Design of Workflows.

[14] C. R. Hofmeister, "Dynamic reconfiguration of distributed applications," 1998.

[15] Y. Simmhan, A. Kumbhare, and C. Wickramaarachchi, "Floe: A dynamic, continusous dataflow framework for elastic clouds," USC, Tech. Rep., 2013.

[16] Q. Zhou, S. Natarajan, Y. Simmhan, and V. Prasanna, "Semantic information modeling for emerging applications in smart grid," in *ITNG*, 2012.

[17] Q. Zhou, Y. Simmhan, and V. Prasanna, "Incorporating semantic knowledge into dynamic data processing for smart power grids," in *The Semantic Web–ISWC 2012*. Springer, 2012.

[18] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, "Pegasus: Mapping scientific workflows onto the grid," in *Grid Computing*. Springer, 2004.

[19] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *SSDBM*. IEEE, 2004.

[20] Y. Simmhan, C. van Ingen, A. S. Szalay, R. S. Barga, and J. Heasley, "Building reliable data pipelines for managing community data using scientific workflows," in *eScience*, 2009.

[21] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat *et al.*, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, 2004.

[22] A. H. H. Ngu, A. Jamnagarwala, G. Chin, C. Sivaramakrishnan, and T. J. Critchlow, "Kepler scientific workflow design and execution with contexts," PNNL, Tech. Rep. PNNL-SA-77613, 2011.

[23] P. Neophytou, P. K. Chrysanthis, and A. Labrinidis, "Towards continuous workflow enactment systems," in *Collaborative Computing: Networking, Applications and Worksharing*. Springer, 2009.

[24] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina *et al.*, "The design of the borealis stream processing engine," in *CIDR*, 2005.

[25] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "Ibm infosphere streams for scalable, real-time, intelligent transportation services," in *SIGMOD*, 2010.

[26] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman, "Dynamic plan migration for continuous queries over data streams," in *SIGMOD*, 2004.

[27] M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen, "The enterprise service bus: making service-oriented architecture real," *IBM Systems Journal*, vol. 44, no. 4, 2005.

GoFFish: A Framework for Distributed Analytics over Timeseries Graphs

Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Nam Ma, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, Viktor Prasanna
University of Southern California
Los Angeles, California 90089 USA
{simmhan, kumbhare, cwickram, namma, snagarka, sathyavi, raghu, prasanna}@usc.edu

ABSTRACT

Massive datasets from scientific instruments and enterprises were the initial Big Data frontiers. But these are being subsumed by complex, high-velocity data from ubiquitous sensors and social network streams. Such datasets are characterized by both temporal attributes and lateral relationships between them forming a graph structure, and scalable data analytics frameworks have not been adequately examined for them. We propose GoFFish, a novel framework for storing and processing timeseries graphs. GoFFish consists of two components: GoFS distributed storage layer for spatially partitioning and storing graphs using temporally co-local file slices, and Gopher subgraph-centric programming model for composing and executing graph analytics on clusters using a BSP-like model. We introduce the GoFFish design and architecture, discuss the synergistic subgraph centric storage and composition model that allow scalable analytics over space and time, and illustrate them using both graph algorithms (connected-components, SSSP) and Big Data applications (Internet traces, social networks) as exemplars.

Keywords

Graph processing, distributed systems, timeseries, distributed storage, programming framework, Big data

1. INTRODUCTION

The well-recognized challenge of "Big Data" manifests itself in many forms, ranging from data collection and storage, to their analysis and visualization. Massive datasets from large scientific instruments (e.g. Large Hadron Collider, Large-Synoptic Sky Survey) and enterprise data warehouses (e.g. web logs, financial transactions) offered the initial grand challenges for data-intensive computing that led to transformative advances such as distributed and NoSQL databases, the MapReduce programming model and datacenters based on commodity hardware. The unprecedented *volume of data* was the dimension of "bigness" that was addressed here.

With the proliferation of ubiquitous physical (e.g. urban monitoring, smart power meters) and virtual (e.g. Twitter feeds, Foursquare check-ins) agents that sense, monitor and track human and environmental activity, data is streaming more continuously and is intrinsically interconnected. Two defining characteristics of such datasets endemic to both the Internet of Things and Social Networks are the temporal or time-series attributes and the lateral relationships that exist between them. Such temporal and graph datasets, in particular datasets that imbue both these features, have not been adequately examined from the perspective of scalable Big Data management and analysis even as they are becoming pervasive.

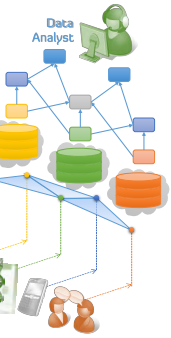
Graph datasets with temporal characteristics have been variously known in literature as temporal graphs [3], kineographs [1], and time-evolving graphs [6]. In this poster, we focus on timeseries graphs which we define to be graphs whose topology is slow-changing but whose properties associated with vertices and edges change (or are generated) more frequently. Many common datasets can be captured using such a time-series graph model. For e.g., when we consider images captured by a network of traffic cameras in a large city, these cameras themselves are inter-connected by the roads that link them (the graph topology), while the periodic data that they generate, such as vehicle license plates, form a time-series graph (Fig. 1a). Each graph instance in the time-series may represent a time instant or range. Such time-series graphs can be constructed from social network feeds (friend network topology with instances formed from tweets) or even Internet traces (Internet IPs and hops form topology while temporal latencies form instances).

The proliferation of "Big Data" has led to the development of both general-purpose and bespoke programming models for composing and executing analytics at scale on Clouds and commodity cluster infrastructure. Hadoop/MapReduce is the obvious example of a generic framework. Despite its popularity and continuous evolution(e.g. Iterative MR [2]), the MapReduce model is better suited for tuple-based dataset operations rather for graph-based datasets. Google's Pregel [4] offers a *vertex-centric* graph programming abstraction that uses a Bulk Synchronous Parallel model with individual tasks being composed as if operating independently on a single vertex. Each barrier synchronization step is called a supersteps and vertices distributed across a cluster can exchange messages with neighbors at these synchronization boundaries. This offers a distributed graph programming model elegant in its simplicity, similar to MapReduce. However, in practice, the barriers are costly and the number of supersteps requires can be large depending on the type of algorithm (e.g. betweenness centrality, All Pairs Shortest Path) [5]. Also, the graph is assumed to be in memory and there is no concept of timeseries

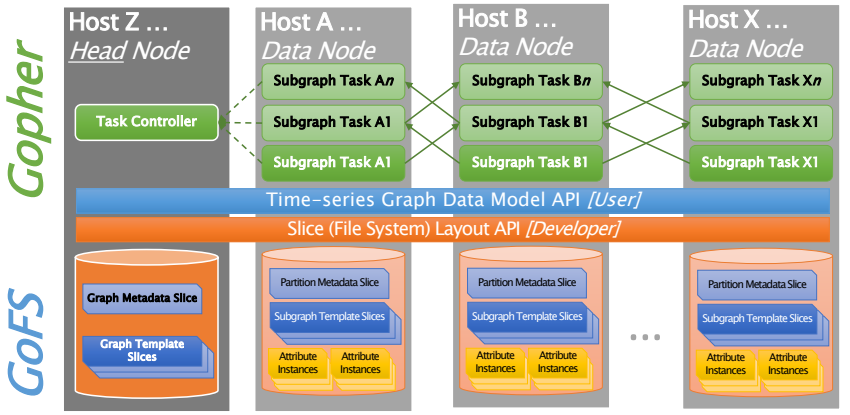
In this poster, we propose GoFFish, a novel framework for stor-

GoFFish: A Framework for Distributed Analytics over Time

Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Nam Ma, Soonil Nagarkar, Santosh Ravi, Cauligi Ragh



Timeseries Graphs



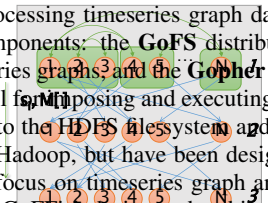
(b) GoFS distributed graph store and Gopher subgraph centric programming model.

Figure 1: Sample scenario where temporal snapshots of sensor data forms timeseries graph and the GoFFish Architecture.

GoFS Distributed Graph Storage

h-Centric Programming Model

ing and processing timeseries graph datasets [1]. GoFFish consists of two components: the **GoFS** distributed storage layer for storing time series graphs, and the **Gopher** *subgraph-centric* programming model for composing and executing graph analytics. These are analogous to the HDFS filesystem and MapReduce programming model for Hadoop, but have been designed and developed *ab initio* with a focus on timeseries graph analytics. We introduce and discuss the GoFFish design and architecture in the next section and offer sample applications – both classic graph algorithms and Big Data applications – in the last section.



Gopher Execution Model

2. GOFFISH ARCHITECTURE

The Graph-oriented File System (GoFS) is a distributed graph store that operates over a network file system. It spatially partitions the graph topology to minimize edge cuts and balance the number vertices per partition, and co-locates all instance data over time for the partition within the same physical host. Further, within a partition (host), it uses a subgraph centric model (where subgraphs are weakly connected components in a partition) for grouping instance data within a single file and further divides the temporal duration into independent "slice" files. This allows collocation of graph instance data for vertices and edges that spatially and temporally

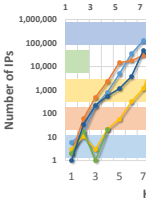
Acknowledgement: This work is supported by a grant from the DARPA XDATA program. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

The Gopher subgraph centric programming model extends the Pregel vertex-centric model by allow application composition for a subgraph at a time. This reduces the number of synchronization supersteps by up to a factor equal to the number of vertices in the subgraph, and there by improves performance while still offering the intuitive programming abstraction. Further, we introduce the notion of temporal message passing across graph instances to compose analytics over timeseries graphs. Gopher intelligently uses the distributed storage model of GoFS to schedule and execute these applications in a cluster environment, minimizing disk I/O costs while also leveraging data parallelism.

3. APPLICATIONS

We show several classic graph algorithms, extended for timeseries graphs, as well a real graph applications operating at scale within the GoFFish framework. These are executed on a 12-node, 96-core cluster environment and compared against an Apache Giraph deployment (implementing Google's Pregel model) running over an HDFS store. Our results show a significant performance improvement for GoFFish.

- Single Source Shortest Path
- Weakly Connected Components
- Path Detection over Space and Time (E.g. License Plate Detection in an urban network of traffic cameras)
- Virtual Internet Trace-routes



4. ACKNOWLEDGMENTS

This work is supported by a grant from the DARPA XDATA program. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. We would like to thank Hsuan-Yi and Da Tong from USC, and other XDATA performers for their assistance

5. REFERENCES

[1] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *ACM Eurosys*, 2012.

[2] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative mapreduce. In *HPDC*, 2010.

[3] V. Kostakos. Temporal graphs. *Physica A*, 2009.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD*, 2010.

[5] M. Redekopp, Y. Simmhan, and V. K. Prasanna. Optimizations and analysis of bsp graph processing models on public clouds. In *IEEE IPDPS*, 2013.

[6] H. Tong, S. Papadimitriou, J. Sun, P. S. Yu, and C. Faloutsos. Colibri: fast mining of large static and dynamic graphs. In *ACM KDD*, 2008.

[1] https://github.com/usc-cloud/goffish